

*code2career*

[Code2Career.ai](https://Code2Career.ai)



## Table of contents

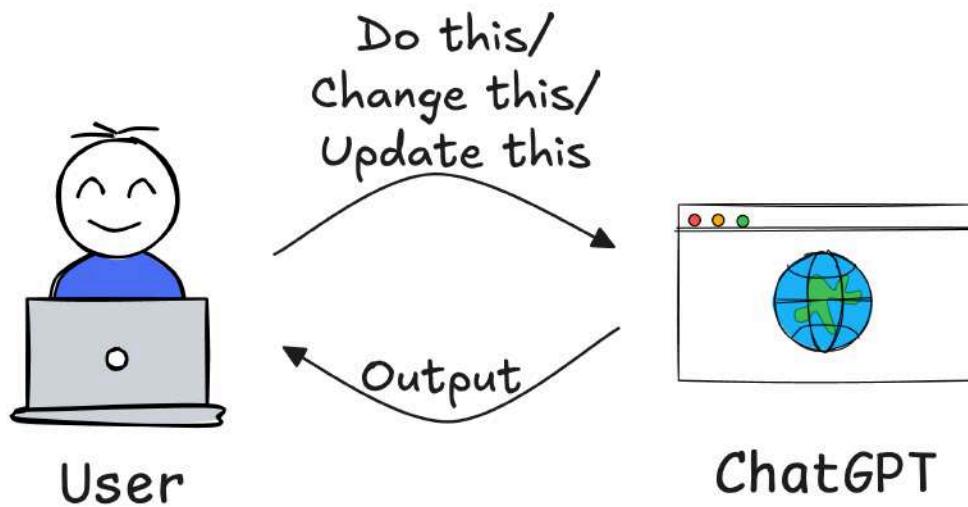
<b>AI Agents.....</b>	<b>4</b>
<b>What is an AI Agent?.....</b>	<b>5</b>
<b>Agent vs LLM vs RAG.....</b>	<b>8</b>
LLM (Large Language Model).....	8
RAG (Retrieval-Augmented Generation).....	9
Agent.....	9
<b>Building blocks of AI Agents.....</b>	<b>10</b>
1) Role-playing.....	10
2) Focus/Tasks.....	11
3) Tools.....	11
#3.1) Custom tools.....	12
#3.2) Custom tools via MCP.....	17
4) Cooperation.....	21
5) Guardrails.....	22
6) Memory.....	22
<b>5 Agentic AI Design Patterns.....</b>	<b>24</b>
#1) Reflection pattern.....	25
#2) Tool use pattern.....	25
#3) ReAct (Reason and Act) pattern.....	26
#4) Planning pattern.....	28
#5) Multi-Agent pattern.....	29
<b>5 Levels of Agentic AI Systems.....</b>	<b>30</b>
#1) Basic responder.....	31
#2) Router pattern.....	31
#3) Tool calling.....	32
#4) Multi-agent pattern.....	32
#5) Autonomous pattern.....	33

<b>AI Agents Projects.....</b>	<b>34</b>
#1) Agentic RAG.....	35
#2) Voice RAG Agent.....	41
#3) Multi-agent Flight finder.....	46
#4) Financial Analyst.....	54
#5) Brand Monitoring System.....	59
#6) Multi-agent Hotel Finder.....	67
#7) Multi-agent Deep Researcher.....	75
#8) Human-like Memory for Agents.....	82
#9) Multi-agent Book Writer.....	89
#10) Multi-agent Content Creation System.....	98
#11) Documentation Writer Flow.....	106
#12) News Generator.....	112

# AI Agents

# What is an AI Agent?

Imagine you want to generate a report on the latest trends in AI research. If you use a standard LLM, you might:

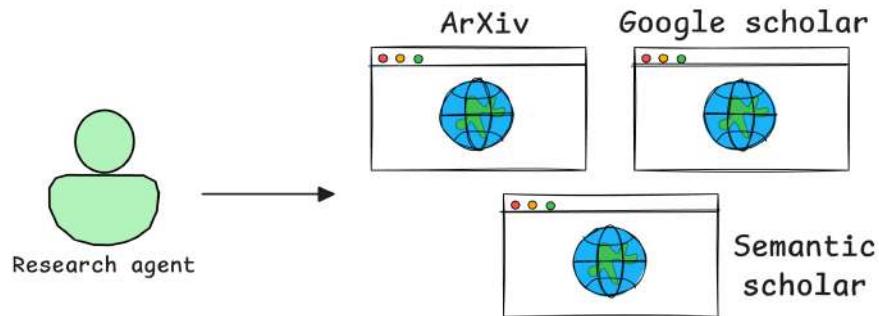


1. Ask for a summary of recent AI research papers.
2. Review the response and realize you need sources.
3. Obtain a list of papers along with citations.
4. Find that some sources are outdated, so you refine your query.
5. Finally, after multiple iterations, you get a useful output.

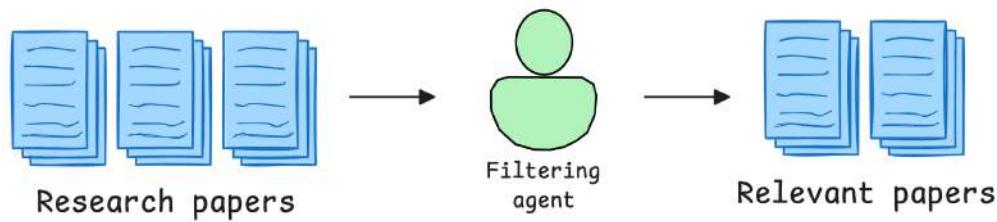
This iterative process takes time and effort, requiring you to act as the decision-maker at every step.

Now, let's see how AI agents handle this differently:

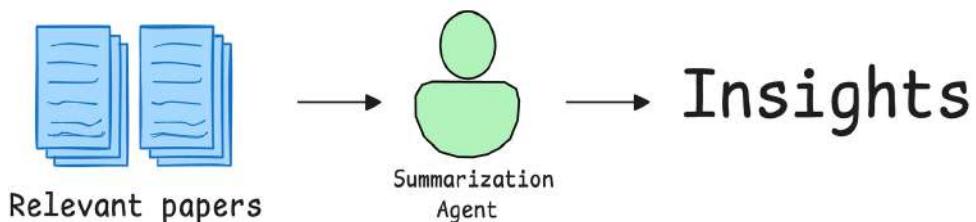
A Research Agent autonomously searches and retrieves relevant AI research papers from arXiv, Semantic Scholar, or Google Scholar.



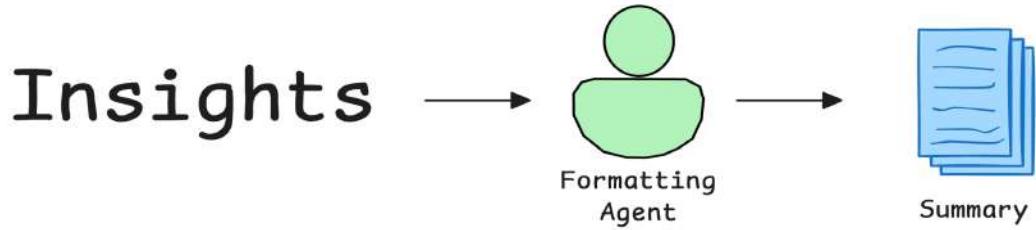
- A Filtering Agent scans the retrieved papers, identifying the most relevant ones based on citation count, publication date, and keywords.



- A Summarization Agent extracts key insights and condenses them into an easy-to-read report.



- A Formatting Agent structures the final report, ensuring it follows a clear, professional layout.



Here, the AI agents not only execute the research process end-to-end but also self-refine their outputs, ensuring the final report is comprehensive, up-to-date, and well-structured - all without requiring human intervention at every step.



To formalize AI Agents are autonomous systems that can reason, think, plan, figure out the relevant sources and extract information from them when needed, take actions, and even correct themselves if something goes wrong.

## Agent vs LLM vs RAG



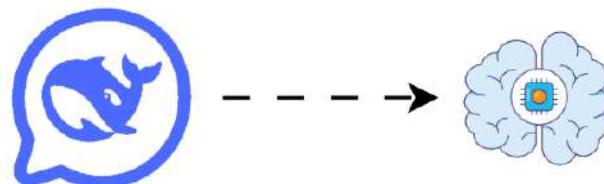
Let's break it down with a simple analogy:

- LLM is the brain.
- RAG is feeding that brain with fresh information.
- An agent is the decision-maker that plans and acts using the brain and the tools.

### LLM (Large Language Model)

An LLM like GPT-4 is trained on massive text data.

It can reason, generate, summarize but only using what it already knows (i.e., its training data).

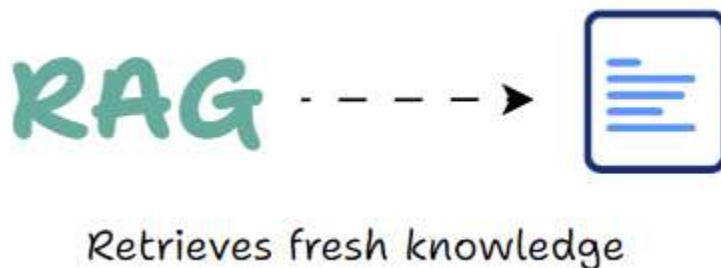


LLM is smart but static

It's smart, but static. It can't access the web, call APIs, or fetch new facts on its own.

## RAG (Retrieval-Augmented Generation)

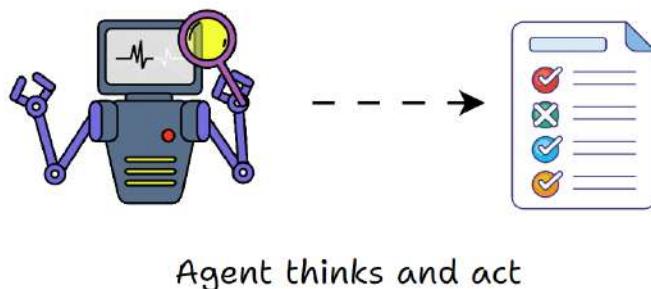
RAG enhances an LLM by retrieving external documents (from a vector DB, search engine, etc.) and feeding them into the LLM as context before generating a response.



RAG makes the LLM aware of updated, relevant info without retraining.

## Agent

An Agent adds autonomy to the mix.



It doesn't just answer a question—it decides what steps to take:

Should it call a tool? Search the web? Summarize? Store info?

An Agent uses an LLM, calls tools, makes decisions, and orchestrates workflows just like a real assistant.

# Building blocks of AI Agents

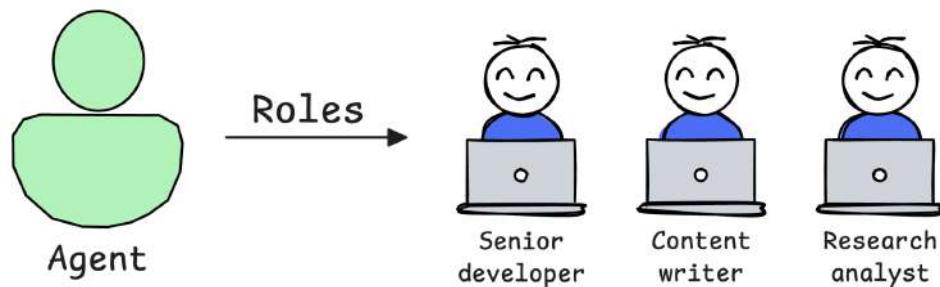
AI agents are designed to reason, plan, and take action autonomously. However, to be effective, they must be built with certain key principles in mind. There are six essential building blocks that make AI agents more reliable, intelligent, and useful in real-world applications:

1. Role-playing
2. Focus
3. Tools
4. Cooperation
5. Guardrails
6. Memory

Let's explore each of these concepts and understand why they are fundamental to building great AI agents.

## 1) Role-playing

One of the simplest ways to boost an agent's performance is by giving it a clear, specific role.



A generic AI assistant may give vague answers. But define it as a "Senior contract lawyer," and it responds with legal precision and context.

Why?

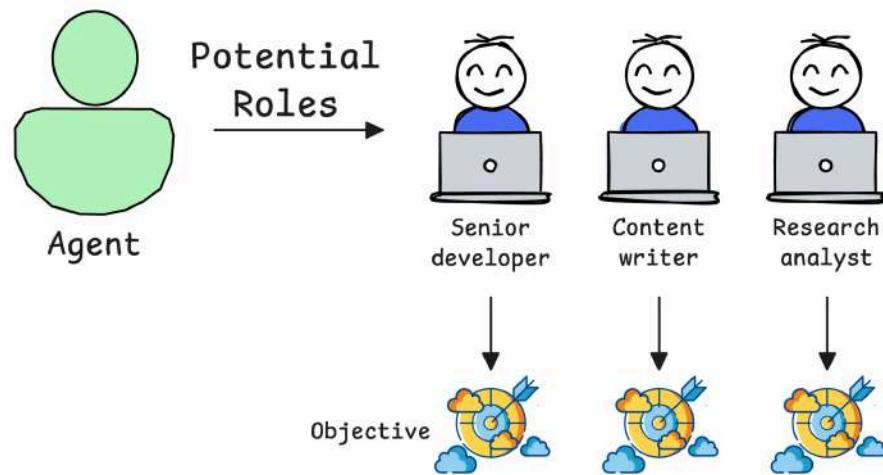
Because role assignment shapes the agent's reasoning and retrieval process. The

more specific the role, the sharper and more relevant the output.

## 2) Focus/Tasks

Focus is key to reducing hallucinations and improving accuracy.

Giving an agent too many tasks or too much data doesn't help - it hurts.



Overloading leads to confusion, inconsistency, and poor results.

For example, a marketing agent should stick to messaging, tone, and audience not pricing or market analysis.

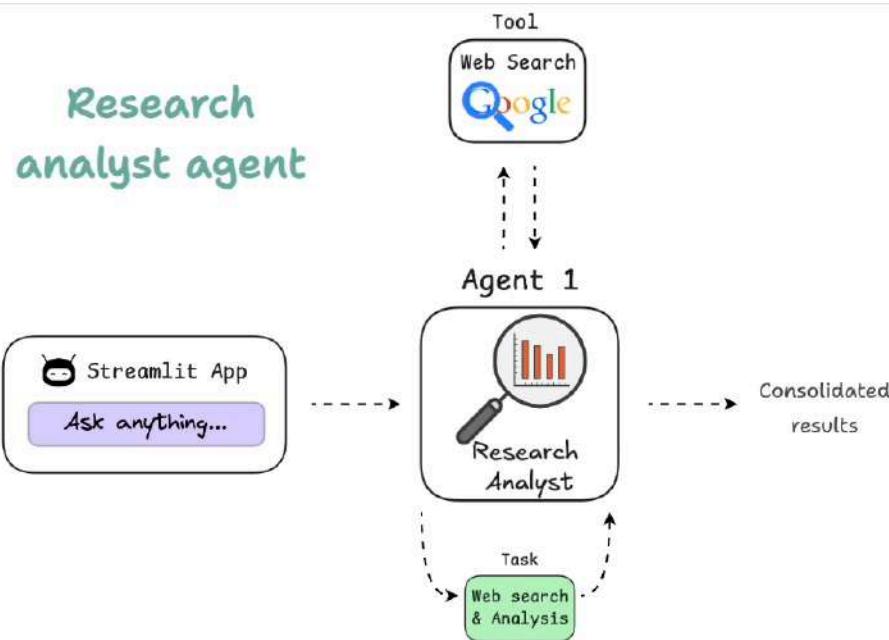
Instead of trying to make one agent do everything, a better approach is to use multiple agents, each with a specific and narrow focus.

Specialized agents perform better - every time.

## 3) Tools

Agents get smarter when they can use the right tools.

But more tools ≠ better results.



For example, an AI research agent could benefit from:

- A web search tool for retrieving recent publications.
- A summarization model for condensing long research papers.
- A citation manager to properly format references.

But if you add unnecessary tools—like a speech-to-text module or a code execution environment—it could confuse the agent and reduce efficiency.

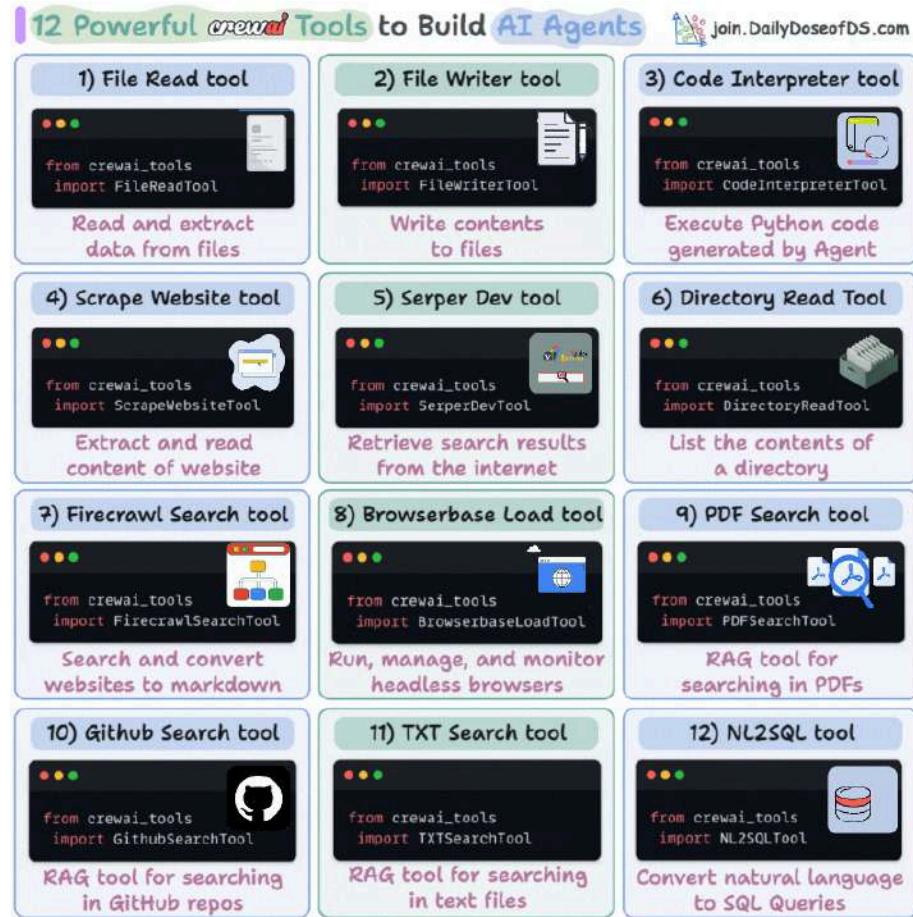
### #3.1) Custom tools

While LLM-powered agents are great at reasoning and generating responses, they lack direct access to real-time information, external systems, and specialized computations.

Tools allow the Agent to:

- Search the web for real-time data.
- Retrieve structured information from APIs and databases.
- Execute code to perform calculations or data transformations.
- Analyze images, PDFs, and documents beyond just text inputs.

CrewAI supports several tools that you can integrate with Agents, as depicted below:

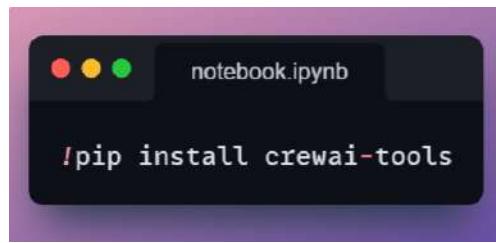


However, you may need to build custom tools at times.

In this example, we're building a real-time currency conversion tool inside CrewAI. Instead of making an LLM guess exchange rates, we integrate a custom tool that fetches live exchange rates from an external API and provides some insights.

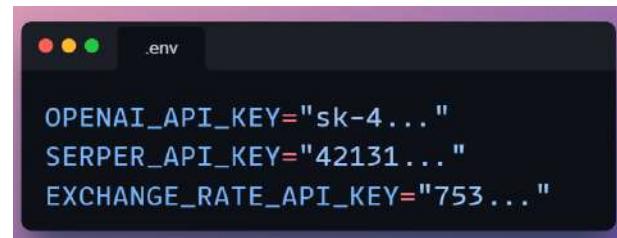
Below, let's look at how you can build one for your custom needs in the CrewAI framework.

Firstly, make sure the tools package is installed:



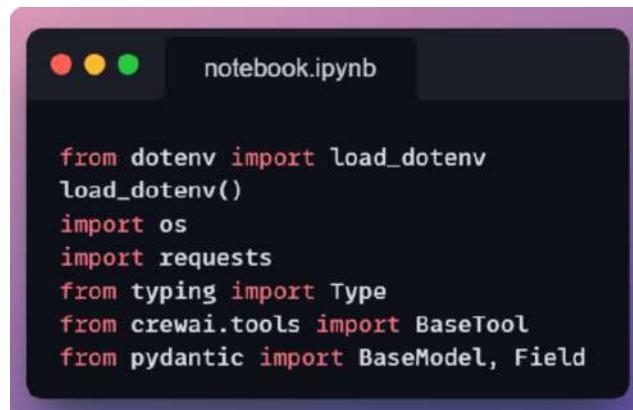
```
!pip install crewai-tools
```

You would also need an API key from here: <https://www.exchangerate-api.com/> (it's free). Specify it in the .env file as shown below:



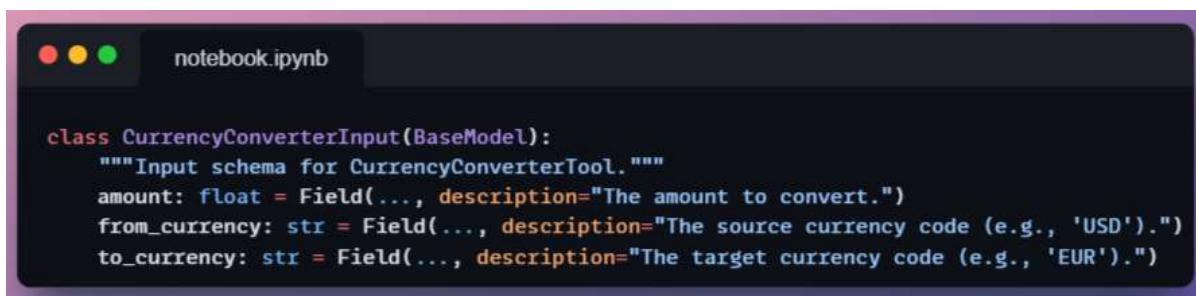
```
OPENAI_API_KEY="sk-4..."  
SERPER_API_KEY="42131..."  
EXCHANGE_RATE_API_KEY="753..."
```

Once that's done, we start with some standard import statements:



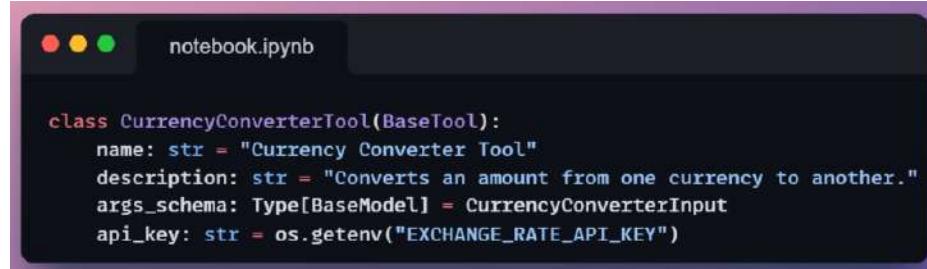
```
from dotenv import load_dotenv  
load_dotenv()  
import os  
import requests  
from typing import Type  
from crewai.tools import BaseTool  
from pydantic import BaseModel, Field
```

Next, we define the input fields the tool expects using Pydantic.



```
class CurrencyConverterInput(BaseModel):  
    """Input schema for CurrencyConverterTool."""  
    amount: float = Field(..., description="The amount to convert.")  
    from_currency: str = Field(..., description="The source currency code (e.g., 'USD').")  
    to_currency: str = Field(..., description="The target currency code (e.g., 'EUR').")
```

Now, we define the CurrencyConverterTool by inheriting from *BaseTool*:

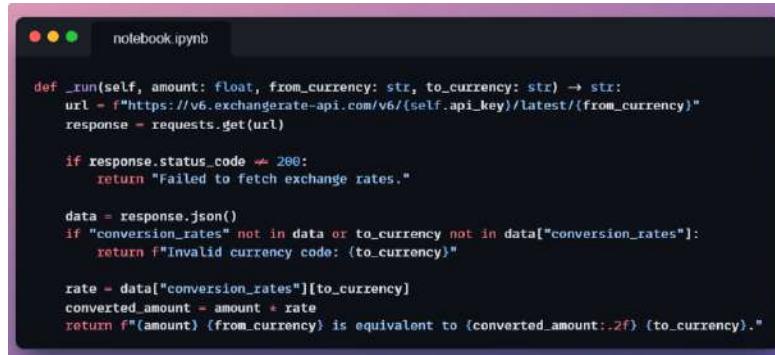


```
notebook.ipynb

class CurrencyConverterTool(BaseTool):
    name: str = "Currency Converter Tool"
    description: str = "Converts an amount from one currency to another."
    args_schema: Type[BaseModel] = CurrencyConverterInput
    api_key: str = os.getenv("EXCHANGE_RATE_API_KEY")
```

Every tool class should have the `_run` method which we will execute whenever the Agents wants to make use of it.

For our use case, we implement it as follows:



```
notebook.ipynb

def _run(self, amount: float, from_currency: str, to_currency: str) -> str:
    url = f"https://v6.exchangerate-api.com/v6/{self.api_key}/latest/{from_currency}"
    response = requests.get(url)

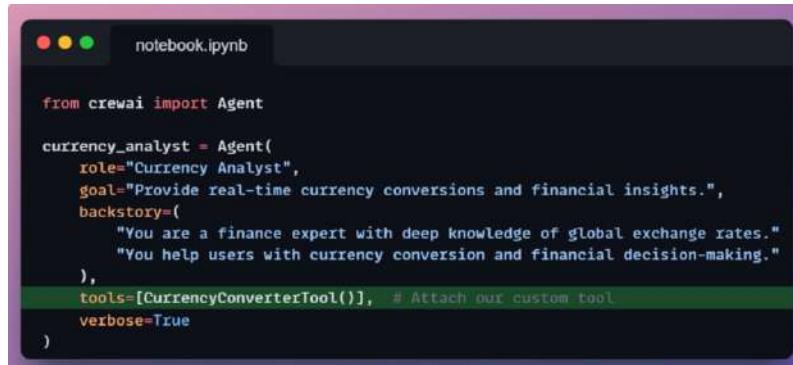
    if response.status_code != 200:
        return "Failed to fetch exchange rates."

    data = response.json()
    if "conversion_rates" not in data or to_currency not in data["conversion_rates"]:
        return f"Invalid currency code: {to_currency}"

    rate = data["conversion_rates"][to_currency]
    converted_amount = amount * rate
    return f"{amount} {from_currency} is equivalent to {converted_amount:.2f} {to_currency}."
```

In the above code, we fetch live exchange rates using an API request. We also handle errors if the request fails or the currency code is invalid.

Now, we define an agent that uses the tool for real-time currency analysis and attach our CurrencyConverterTool, allowing the agent to call it directly if needed:

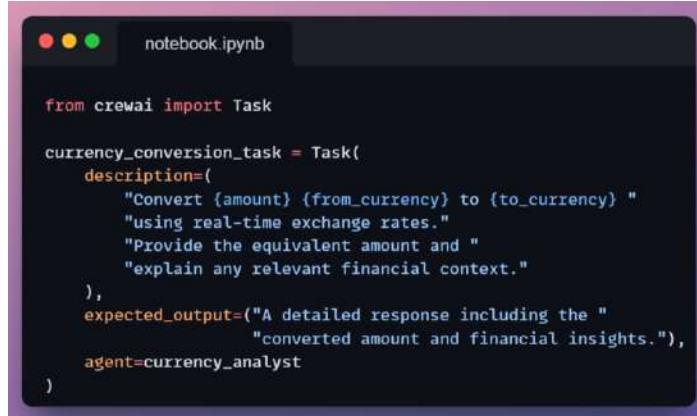


```
notebook.ipynb

from crewai import Agent

currency_analyst = Agent(
    role="Currency Analyst",
    goal="Provide real-time currency conversions and financial insights.",
    backstory=(
        "You are a finance expert with deep knowledge of global exchange rates."
        "You help users with currency conversion and financial decision-making."
    ),
    tools=[CurrencyConverterTool()], # Attach our custom tool
    verbose=True
)
```

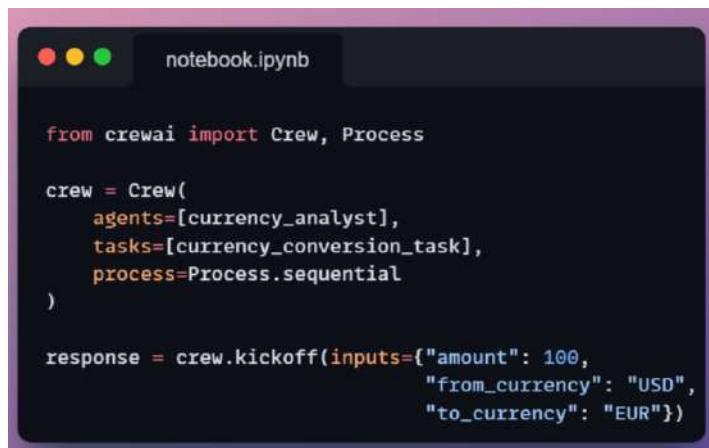
We assign a task to the currency\_analyst agent.



```
from crewai import Task

currency_conversion_task = Task(
    description=(
        "Convert {amount} {from_currency} to {to_currency} using real-time exchange rates."
        "Provide the equivalent amount and explain any relevant financial context."
    ),
    expected_output="A detailed response including the converted amount and financial insights.",
    agent=currency_analyst
)
```

Finally, we create a Crew, assign the agent to the task, and execute it.

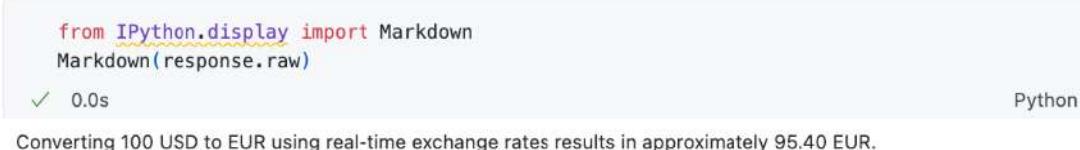


```
from crewai import Crew, Process

crew = Crew(
    agents=[currency_analyst],
    tasks=[currency_conversion_task],
    process=Process.sequential
)

response = crew.kickoff(inputs={"amount": 100,
                                 "from_currency": "USD",
                                 "to_currency": "EUR"})
```

Printing the response, we get the following output:



```
from IPython.display import Markdown
Markdown(response.raw)
```

✓ 0.0s Python

Converting 100 USD to EUR using real-time exchange rates results in approximately 95.40 EUR.

In the financial context, it's worth noting that exchange rates can fluctuate due to various factors like economic indicators, interest rates, and geopolitical events. As of now, the conversion reflects current market conditions, which are influenced by the latest economic data releases and monetary policies in both the United States and the Eurozone. Given the recent trends, if you're planning a trip to Europe or making an investment, these rates may change, so it's beneficial to monitor them regularly.

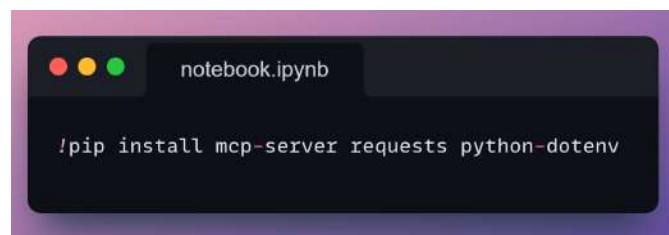
Works as expected!

## #3.2) Custom tools via MCP

Now, let's take it a step further.

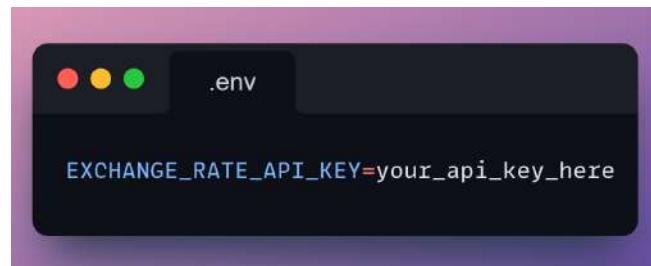
Instead of embedding the tool directly in every Crew, we'll expose it as a reusable MCP tool—making it accessible across multiple agents and flows via a simple server.

First, install the required packages:



```
/pip install mcp-server requests python-dotenv
```

We'll continue using ExchangeRate-API in our .env file:



```
EXCHANGE_RATE_API_KEY=your_api_key_here
```

We'll now write a lightweight server.py script that exposes the currency converter tool. We start with the standard imports:



```
import requests, os
from dotenv import load_dotenv
from mcp.server.fastmcp import FastMCP
```

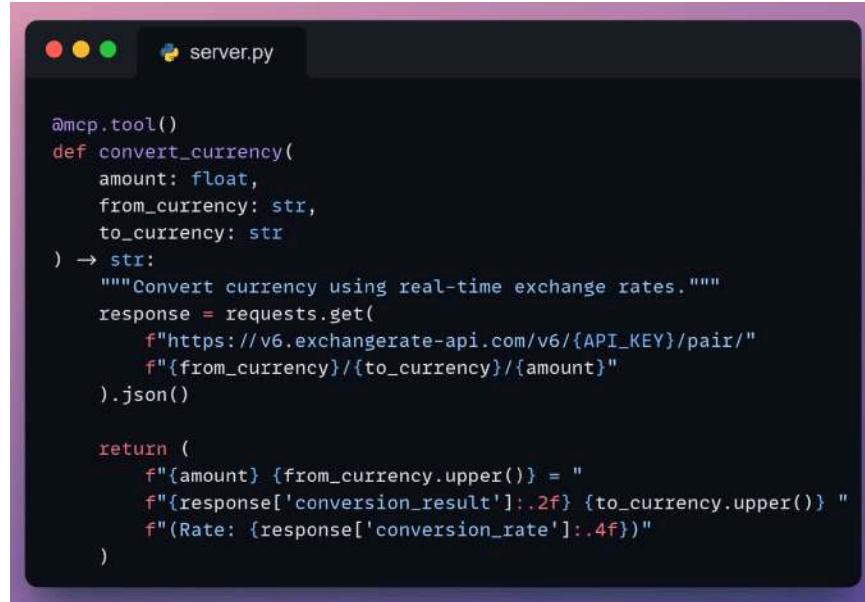
Now, we load environment variables and initialize the server:



```
load_dotenv()

mcp = FastMCP('currency-converter-server', port=8081)
API_KEY = os.getenv("EXCHANGE_RATE_API_KEY")
```

Next, we define the tool logic with `@mcp.tool()`:



```
@mcp.tool()
def convert_currency(
    amount: float,
    from_currency: str,
    to_currency: str
) -> str:
    """Convert currency using real-time exchange rates."""
    response = requests.get(
        f"https://v6.exchangerate-api.com/v6/{API_KEY}/pair/"
        f"{from_currency}/{to_currency}/{amount}"
    ).json()

    return (
        f"{amount} {from_currency.upper()} = "
        f"{response['conversion_result']:.2f} {to_currency.upper()} "
        f"(Rate: {response['conversion_rate']:.4f})"
    )
```

This function takes three inputs—amount, source currency, and target currency—and returns the converted result using the real-time exchange rate API.

To make the tool accessible, we need to run the MCP server. Add this at the end of your script:

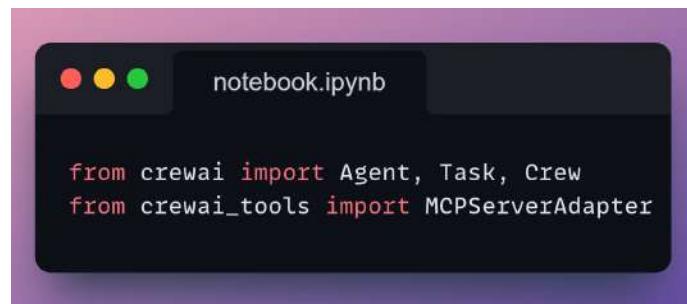


```
if __name__ == "__main__":
    mcp.run(transport="sse")
```

This starts the server and exposes your convert\_currency tool at:  
<http://localhost:8081/sse>.

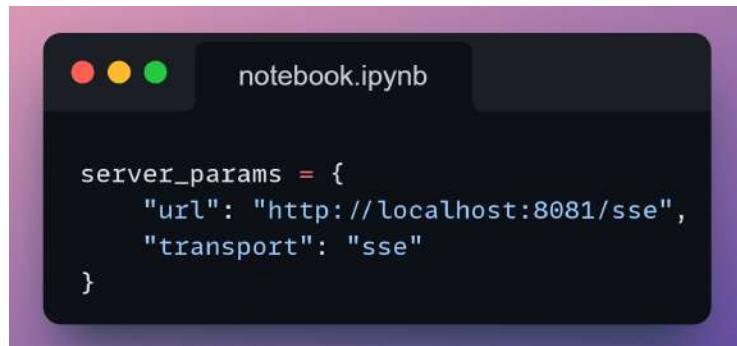
Now any CrewAI agent can connect to it using MCPServerAdapter. Let's now consume this tool from within a CrewAI agent.

First, we import the required CrewAI classes. We'll use Agent, Task, and Crew from CrewAI, and MCPServerAdapter to connect to our tool server.



```
from crewai import Agent, Task, Crew
from crewai_tools import MCPServerAdapter
```

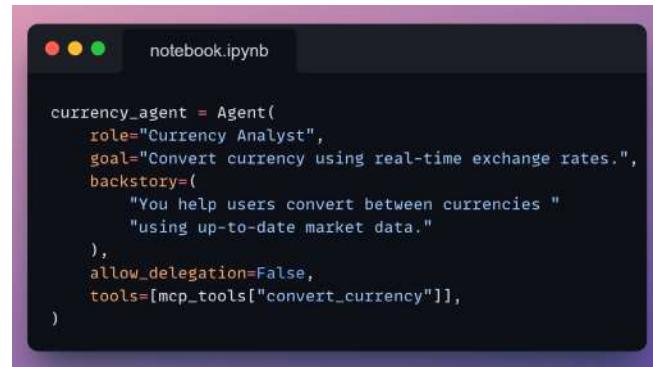
Next, we connect to the MCP tool server. Define the server parameters to connect to your running tool (from server.py).



```
server_params = {
    "url": "http://localhost:8081/sse",
    "transport": "sse"
}
```

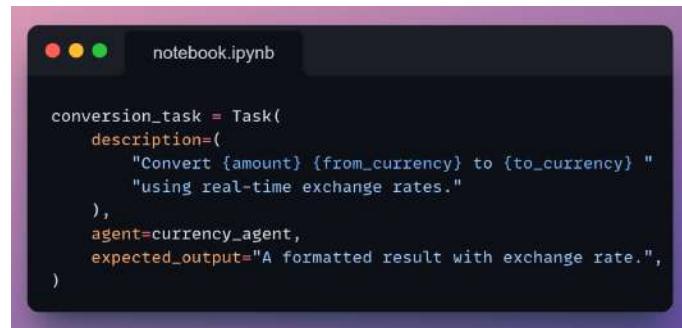
Now, we use the discovered MCP tool in an agent:

This agent is assigned the convert\_currency tool from the remote server. It can now call the tool just like a locally defined one.



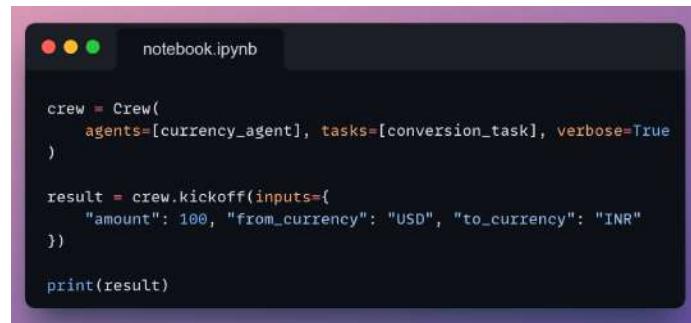
```
currency_agent = Agent(
    role="Currency Analyst",
    goal="Convert currency using real-time exchange rates.",
    backstory=(
        "You help users convert between currencies "
        "using up-to-date market data."
    ),
    allow_delegation=False,
    tools=[mcp_tools["convert_currency"]],
)
```

We give the agent a task description:



```
conversion_task = Task(
    description=(
        "Convert {amount} {from_currency} to {to_currency} "
        "using real-time exchange rates."
    ),
    agent=currency_agent,
    expected_output="A formatted result with exchange rate."
)
```

Finally, we create the Crew, pass in the inputs and run it:

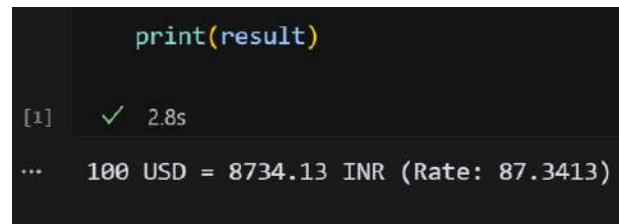


```
crew = Crew(
    agents=[currency_agent], tasks=[conversion_task], verbose=True
)

result = crew.kickoff(inputs={
    "amount": 100, "from_currency": "USD", "to_currency": "INR"
})

print(result)
```

Printing the result, we get the following output:



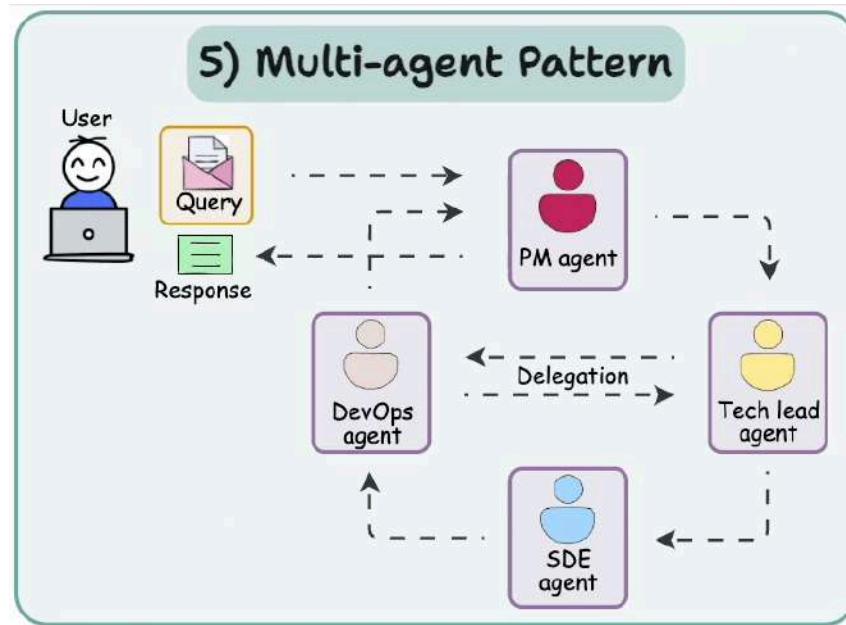
```
print(result)

[1] ✓ 2.8s
...
... 100 USD = 8734.13 INR (Rate: 87.3413)
```

## 4) Cooperation

Multi-agent systems work best when agents collaborate and exchange feedback.

Instead of one agent doing everything, a team of specialized agents can split tasks and improve each other's outputs.



Consider an AI-powered financial analysis system:

- One agent gathers data
- another assesses risk,
- a third builds strategy,
- and a fourth writes the report

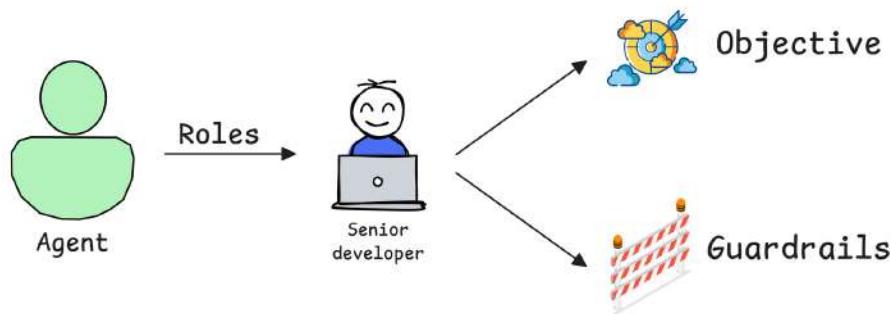
Collaboration leads to smarter, more accurate results.

The best practice is to enable agent collaboration by designing workflows where agents can exchange insights and refine their responses together.

## 5) Guardrails

Agents are powerful but without constraints, they can go off track. They might hallucinate, loop endlessly, or make bad calls.

Guardrails ensure that agents stay on track and maintain quality standards.



Examples of useful guardrails include:

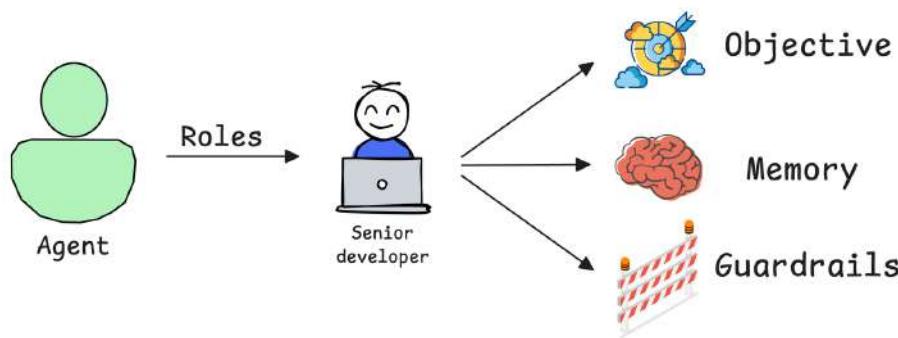
- Limiting tool usage: Prevent an agent from overusing APIs or generating irrelevant queries.
- Setting validation checkpoints: Ensure outputs meet predefined criteria before moving to the next step.
- Establishing fallback mechanisms: If an agent fails to complete a task, another agent or human reviewer can intervene.

For example, an AI-powered legal assistant should avoid outdated laws or false claims - guardrails ensure that.

## 6) Memory

Finally, we have memory, which is one of the most critical components of AI agents.

Without memory, an agent would start fresh every time, losing all context from previous interactions. With memory, agents can improve over time, remember past actions, and create more cohesive responses.



Different types of memory in AI agents include:

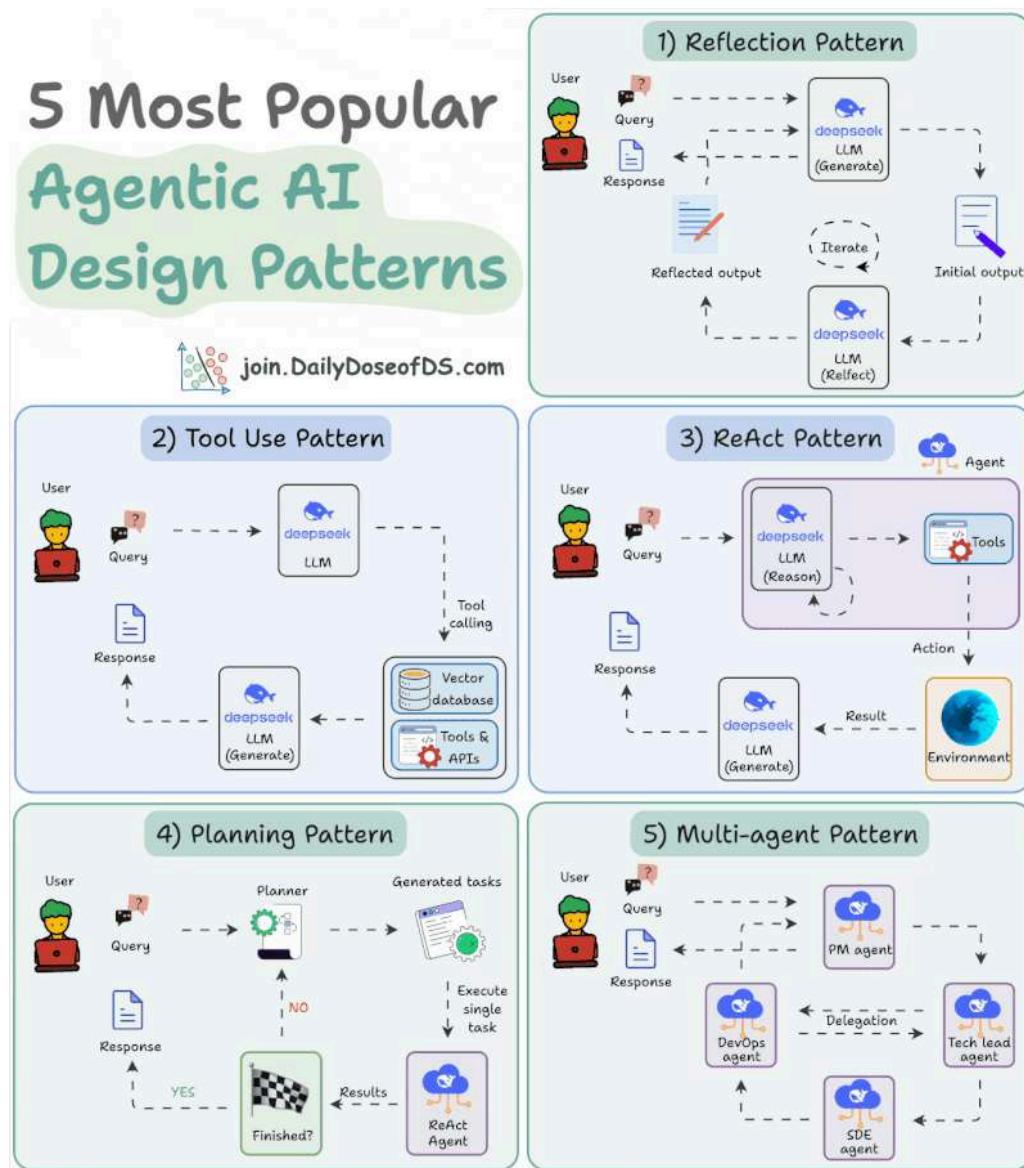
- Short-term memory – Exists only during execution (e.g., recalling recent conversation history).
- Long-term memory – Persists after execution (e.g., remembering user preferences over multiple interactions).
- Entity memory – Stores information about key subjects discussed (e.g., tracking customer details in a CRM agent).

For example, in an AI-powered tutoring system, memory allows the agent to recall past lessons, tailor feedback, and avoid repetition.

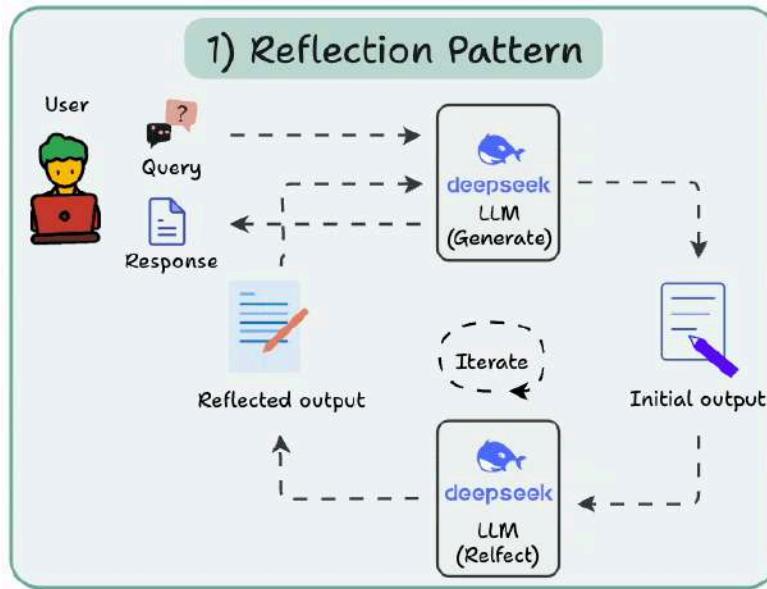
# 5 Agentic AI Design Patterns

Agentic behaviors allow LLMs to refine their output by incorporating self-evaluation, planning, and collaboration!

The following visual depicts the 5 most popular design patterns employed in building AI agents.

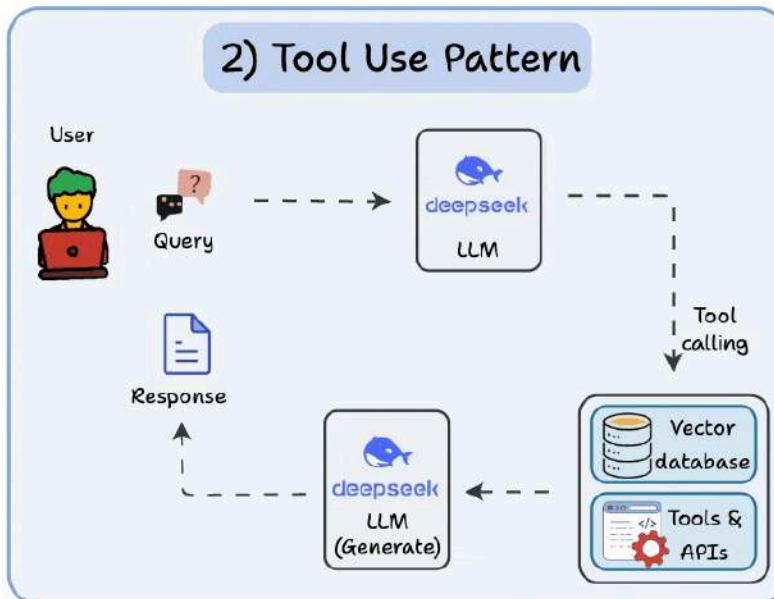


## #1) Reflection pattern



The AI reviews its own work to spot mistakes and iterate until it produces the final response.

## #2) Tool use pattern

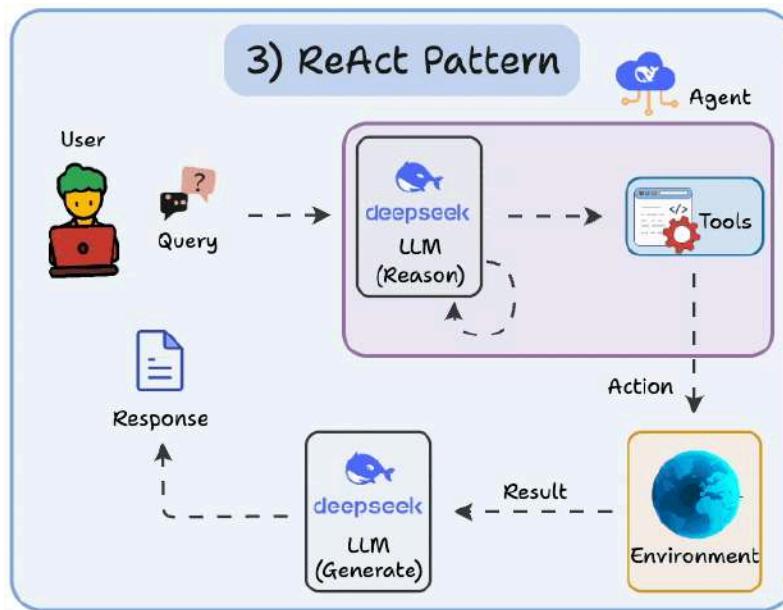


Tools allow LLMs to gather more information by:

- Querying a vector database
- Executing Python scripts
- Invoking APIs, etc.

This is helpful since the LLM is not solely reliant on its internal knowledge.

### #3) ReAct (Reason and Act) pattern

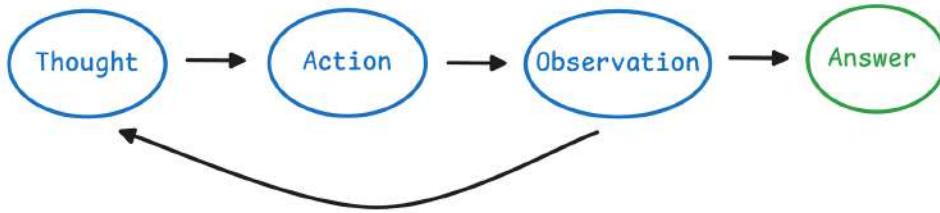


ReAct combines the above two patterns:

- The Agent reflects on the generated outputs.
- It interacts with the world using tools.

A ReAct agent operates in a loop of Thought → Action → Observation, repeating until it reaches a solution or a final answer. This is analogous to how humans solve problems:

## ReAct Pattern



*Note: Frameworks like CrewAI primarily use this by default.*

To understand this, consider the output of a multi-agent system below:

```

# Agent: News Collector
## Task: Search for the latest news on Agent2Agent Protocol

# Agent: News Collector
## Final Answer:
1. **"Agent2Agent Protocol Expands Blockchain Communication"** - November 10, 2023**
   The Agent2Agent Protocol has announced its latest advancements in facilitating seamless communication between decentralized agents on various blockchain networks.

2. **"Agent2Agent Announces Strategic Partnership"** - November 7, 2023**
   A significant partnership has been formed between Agent2Agent Protocol and a leading technology firm to broaden their capabilities in artificial intelligence and machine learning.

# Agent: News Reporter
## Task: Create a news headline on Agent2Agent Protocol from the News Collector. Also validate the news is relevant to Agent2Agent Protocol using the internet

# Agent: News Reporter
## Thought: I need to validate the relevance of the news related to the Agent2Agent Protocol by searching the internet for recent developments.
## Using tool: Search the Internet
## Tool Input:
"{"search_query": \"Agent2Agent Protocol news November 2023\"}"
## Tool Output:
Search results: Title: Announcing the Agent2Agent Protocol (A2A)
Link: https://developers.googleblog.com/2023/a-new-era-of-agent-interoperability/
Snippet: Today, we're launching a new, open protocol called Agent2Agent (A2A), with support and contributions from more than 50 technology partners

# Agent: News Reporter
## Thought: The search results indicate that the Agent2Agent Protocol is currently being discussed in various contexts, especially in relation to its recent announcements and partnerships.
## Using tool: Search the Internet
## Tool Input:
"{"search_query": \"Agent2Agent Protocol latest news\"}"
## Tool Output:
Search results: Title: Announcing the Agent2Agent Protocol (A2A)
Link: https://developers.googleblog.com/2023/a-new-era-of-agent-interoperability/
Snippet: Today, we're launching a new, open protocol called Agent2Agent (A2A), with support and contributions from more than 50 technology partners

# Agent: News Reporter
## Thought: The Agent2Agent Protocol has made significant strides recently with new announcements and partnerships involving AI and blockchain technology.
## Using tool: Search the Internet
## Tool Input:
"{"search_query": \"Agent2Agent Protocol news updates\"}"
## Tool Output:
Search results: Title: Announcing the Agent2Agent Protocol (A2A)
Link: https://developers.googleblog.com/2023/a-new-era-of-agent-interoperability/
Snippet: Today, we're launching a new, open protocol called Agent2Agent (A2A), with support and contributions from more than 50 technology partners
  
```

As shown above, the Agent is going through a series of thought activities before producing a response.

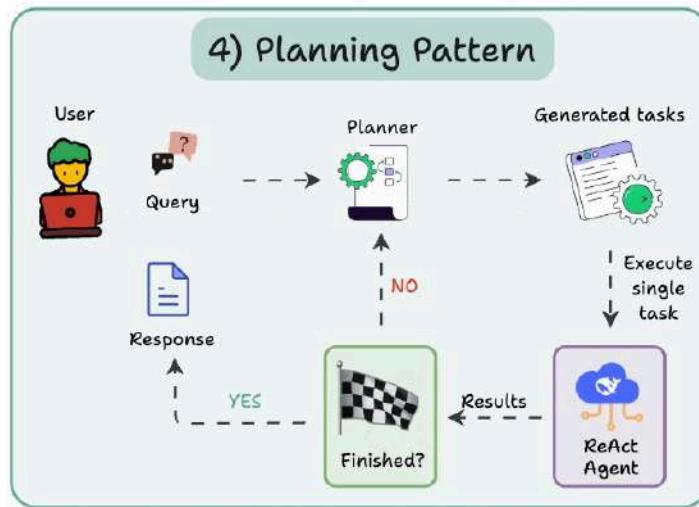
This is the ReAct pattern in action!

More specifically, under the hood, many such frameworks use the ReAct (Reasoning and Acting) pattern to let LLM think through problems and use tools to act on the world.

For example, an agent in CrewAI typically alternates between reasoning about a task and acting (using a tool) to gather information or execute steps, following the ReAct paradigm.

This enhances an LLM agent's ability to handle complex tasks and decisions by combining chain-of-thought reasoning with external tool use like in this [ReAct implementation from scratch](#).

## #4) Planning pattern



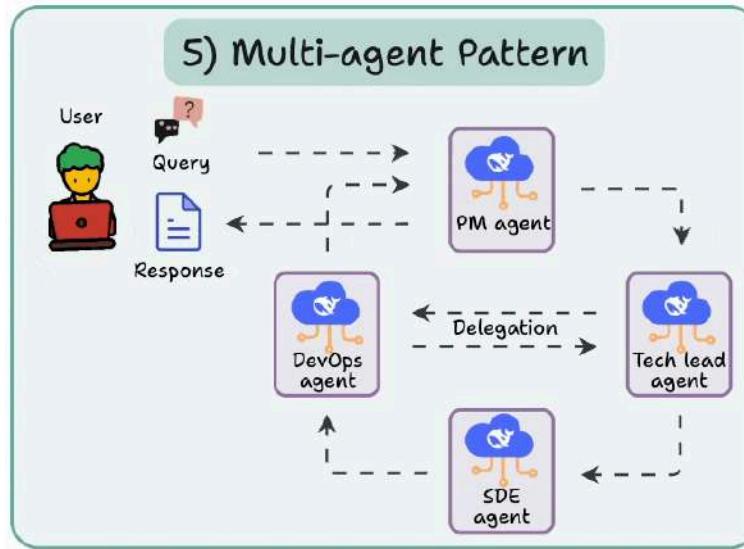
Instead of solving a task in one go, the AI creates a roadmap by:

- Subdividing tasks
- Outlining objectives

This strategic thinking solves tasks more effectively.

*Note: In CrewAI, specify `planning=True` to use Planning.*

## #5) Multi-Agent pattern



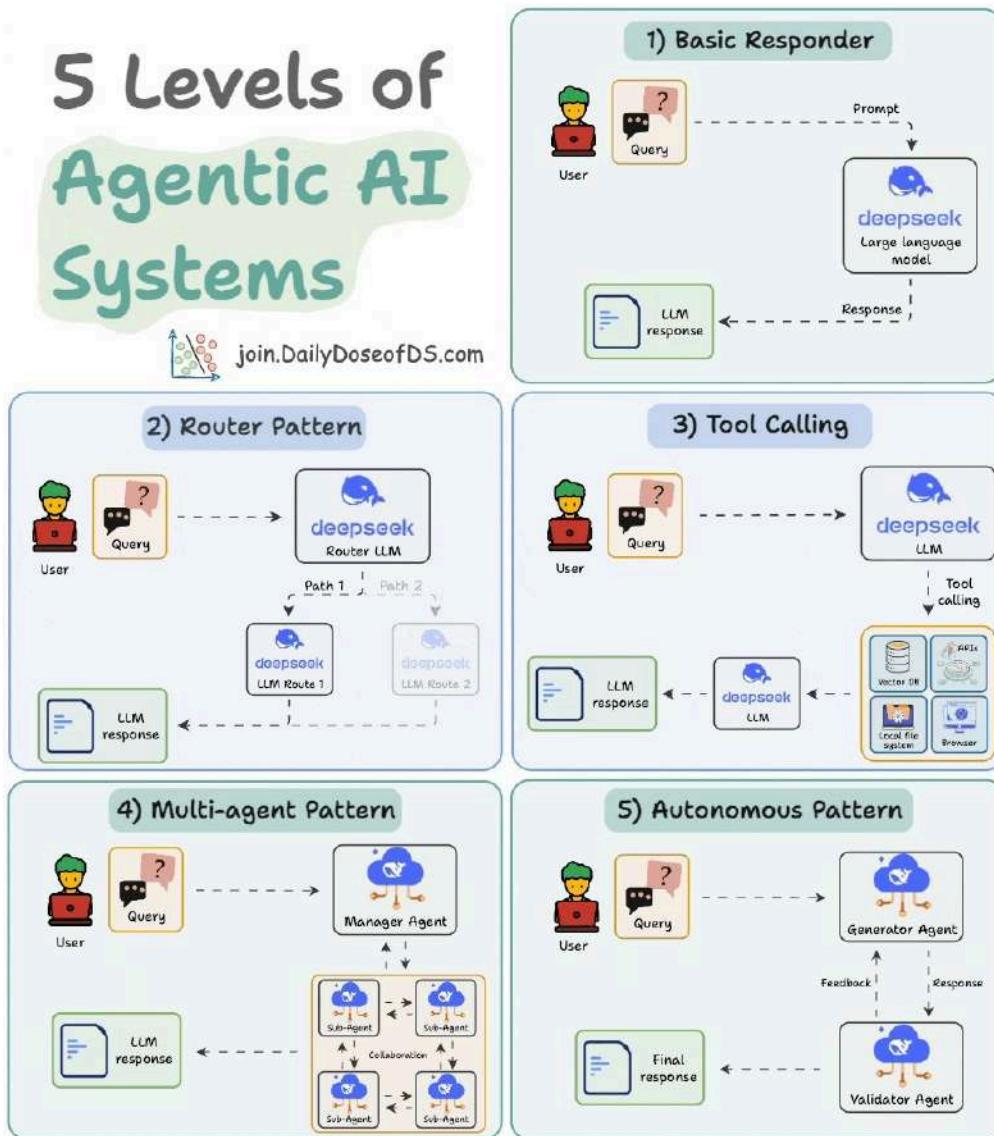
- There are several agents, each with a specific role and task.
- Each agent can also access tools.

All agents work together to deliver the final outcome, while delegating tasks to other agents if needed.

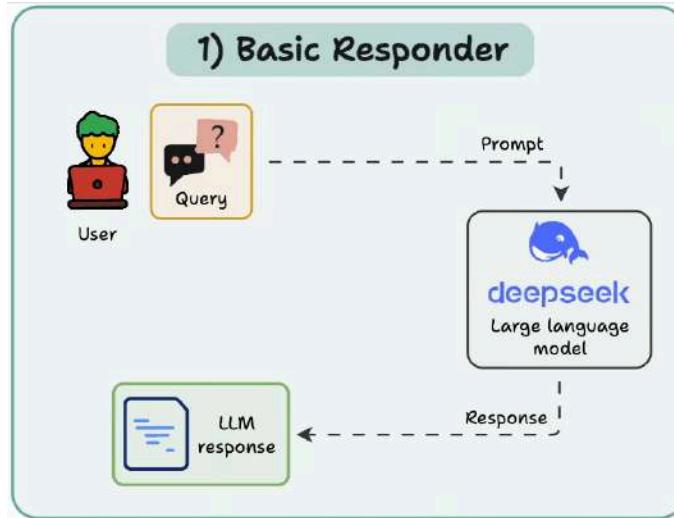
# 5 Levels of Agentic AI Systems

Agentic AI systems don't just generate text; they can make decisions, call functions, and even run autonomous workflows.

The visual explains 5 levels of AI agency—from simple responders to fully autonomous agents.



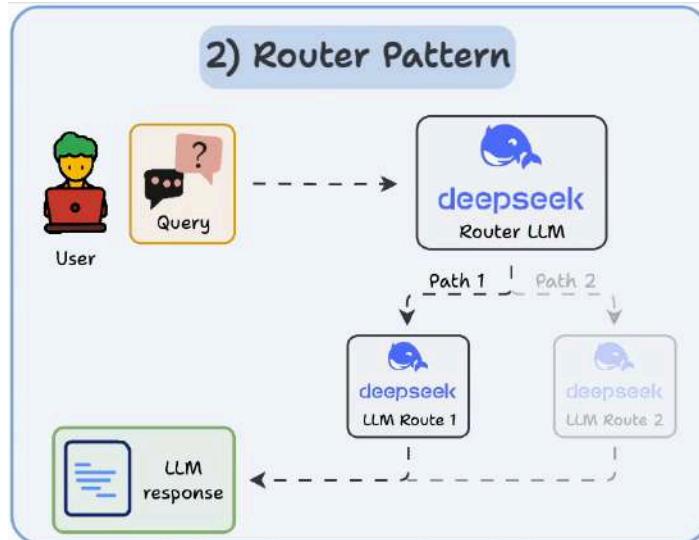
## #1) Basic responder



A human guides the entire flow.

The LLM is just a generic responder that receives an input and produces an output. It has little control over the program flow.

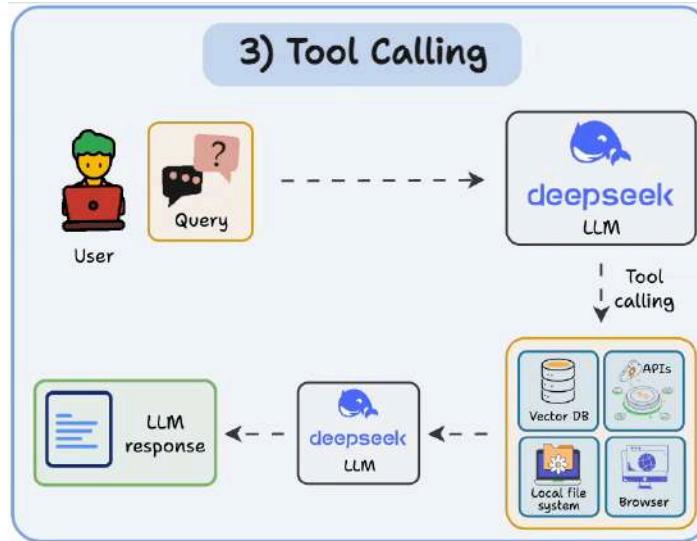
## #2) Router pattern



A human defines the paths/functions that exist in the flow.

The LLM makes basic decisions on which function or path it can take.

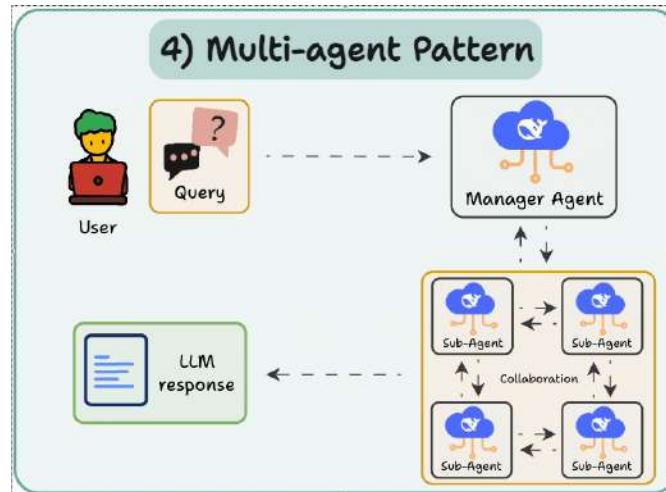
### #3) Tool calling



A human defines a set of tools the LLM can access to complete a task.

LLM decides when to use them and also the arguments for execution.

### #4) Multi-agent pattern

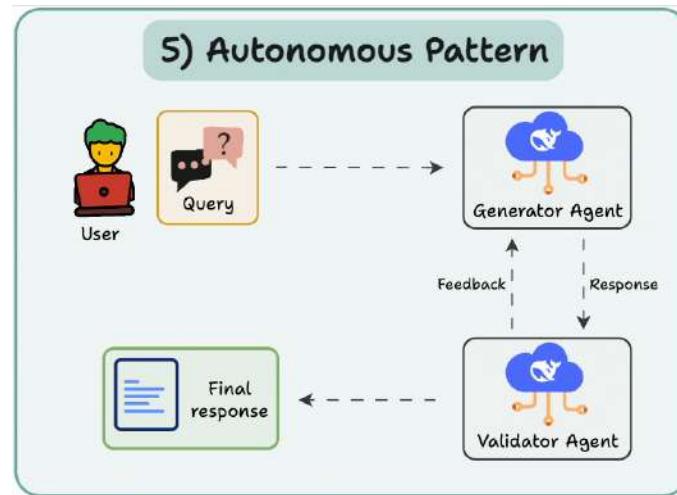


A manager agent coordinates multiple sub-agents and decides the next steps iteratively.

A human lays out the hierarchy between agents, their roles, tools, etc.

The LLM controls execution flow, deciding what to do next.

## #5) Autonomous pattern

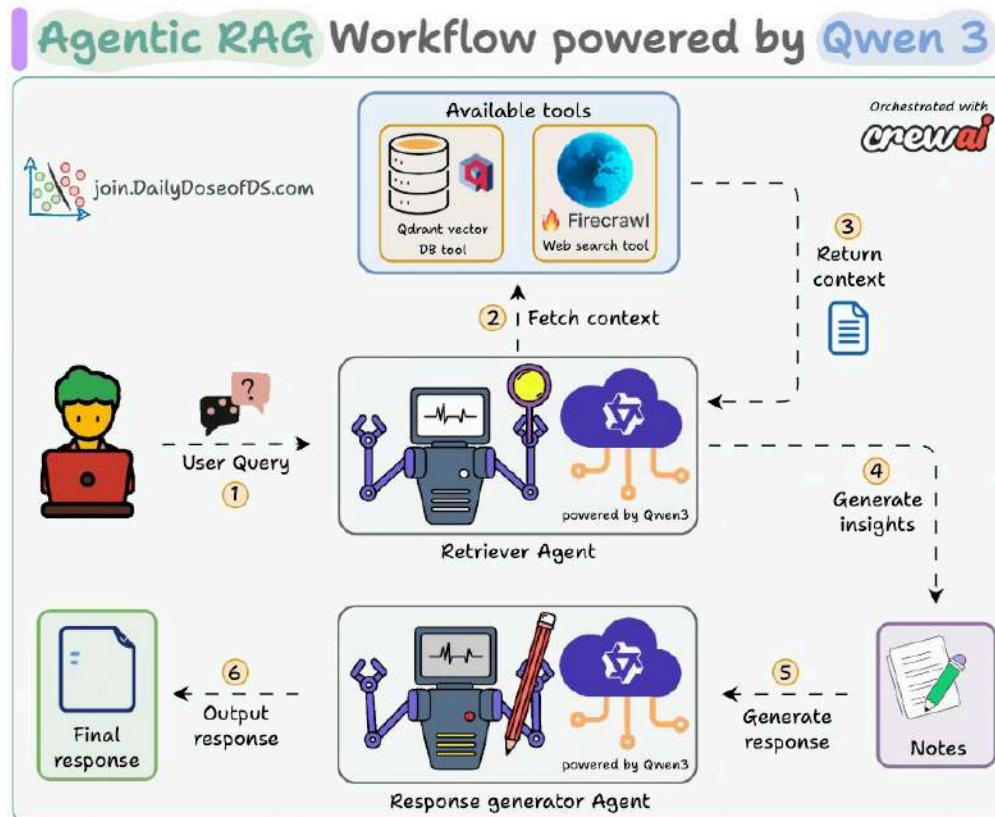


The most advanced pattern, wherein, the LLM generates and executes new code independently, effectively acting as an independent AI developer.

# AI Agents Projects

## #1) Agentic RAG

Build a RAG pipeline with agentic capabilities that can dynamically fetch context from different sources, like a vector DB and the internet.



Tech stack:

- CrewAI for Agent orchestration.
- Firecrawl for web search.
- LightningAI's LitServe for deployment.

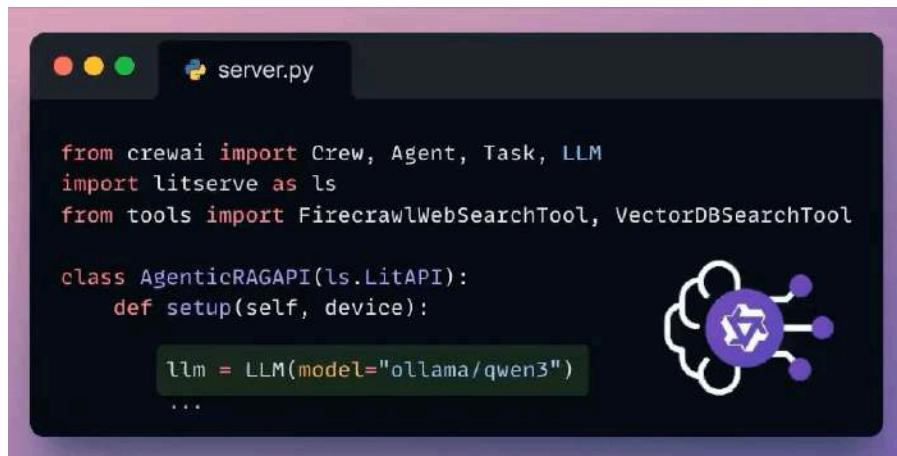
Workflow:

- The Retriever Agent accepts the user query.
- It invokes a relevant tool (Firecrawl web search or vector DB tool) to get context and generate insights.
- The Writer Agent generates a response.

Let's implement it!

### #1) Set up LLM

CrewAI seamlessly integrates with all popular LLMs and providers. Here's how we set up a local Qwen 3 via Ollama:



```
from crewai import Crew, Agent, Task, LLM
import litserve as ls
from tools import FirecrawlWebSearchTool, VectorDBSearchTool

class AgenticRAGAPI(ls.LitAPI):
    def setup(self, device):

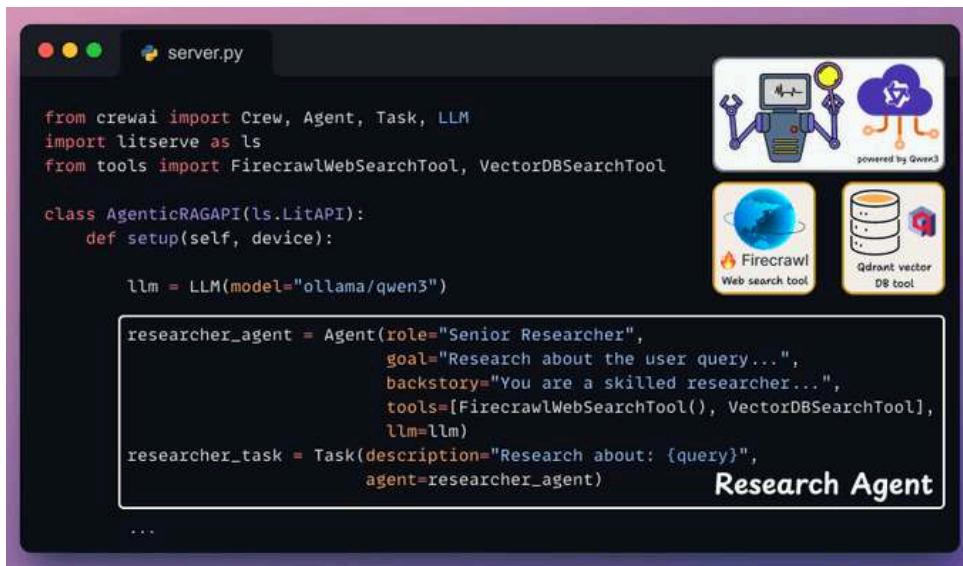
        llm = LLM(model="ollama/qwen3")
        ...

```

### #2) Define Research Agent and Task

This Agent accepts the user query and retrieves the relevant context using a vectorDB tool and a web search tool powered by Firecrawl.

Again, put this in the LitServe setup() method:



```
from crewai import Crew, Agent, Task, LLM
import litserve as ls
from tools import FirecrawlWebSearchTool, VectorDBSearchTool

class AgenticRAGAPI(ls.LitAPI):
    def setup(self, device):

        llm = LLM(model="ollama/qwen3")

        researcher_agent = Agent(role="Senior Researcher",
                                goal="Research about the user query...",
                                backstory="You are a skilled researcher...",
                                tools=[FirecrawlWebSearchTool(), VectorDBSearchTool],
                                llm=llm)
        researcher_task = Task(description="Research about: {query}",
                               agent=researcher_agent)

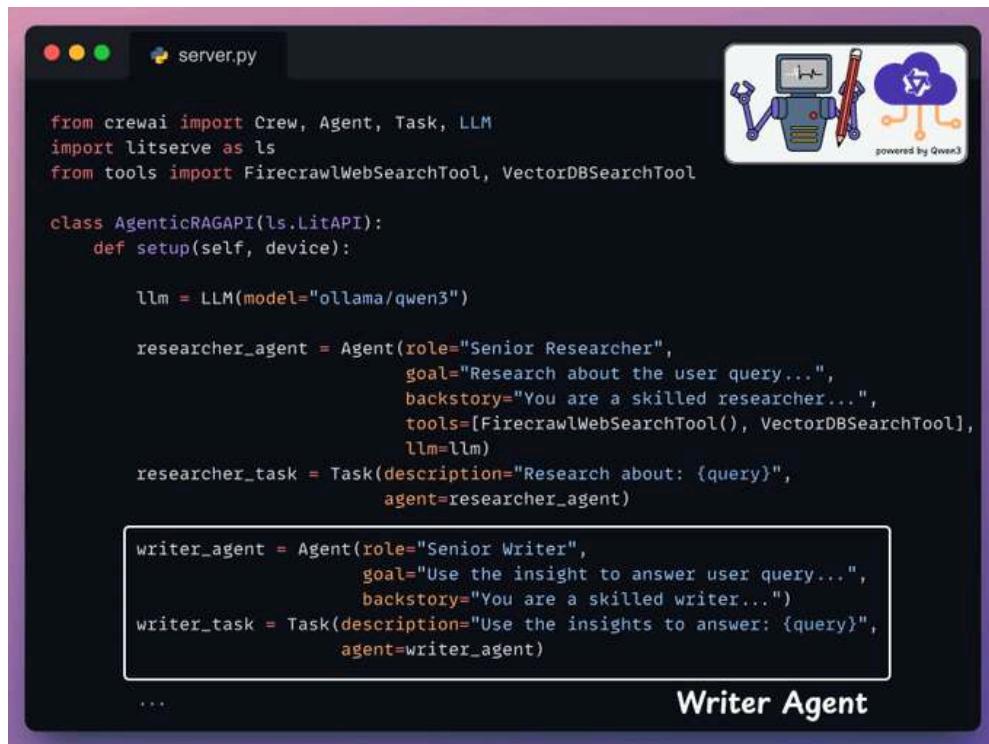
        ...

```

### #3) Define Writer Agent and Task

Next, the Writer Agent accepts the insights from the Researcher Agent to generate a response.

Yet again, we add this in the LitServe setup method:



```
server.py
```

```
from crewai import Crew, Agent, Task, LLM
import litserve as ls
from tools import FirecrawlWebSearchTool, VectorDBSearchTool

class AgenticRAGAPI(ls.LitAPI):
    def setup(self, device):

        llm = LLM(model="ollama/qwen3")

        researcher_agent = Agent(role="Senior Researcher",
                                goal="Research about the user query...",
                                backstory="You are a skilled researcher...",
                                tools=[FirecrawlWebSearchTool(), VectorDBSearchTool],
                                llm=llm)
        researcher_task = Task(description="Research about: {query}",
                               agent=researcher_agent)

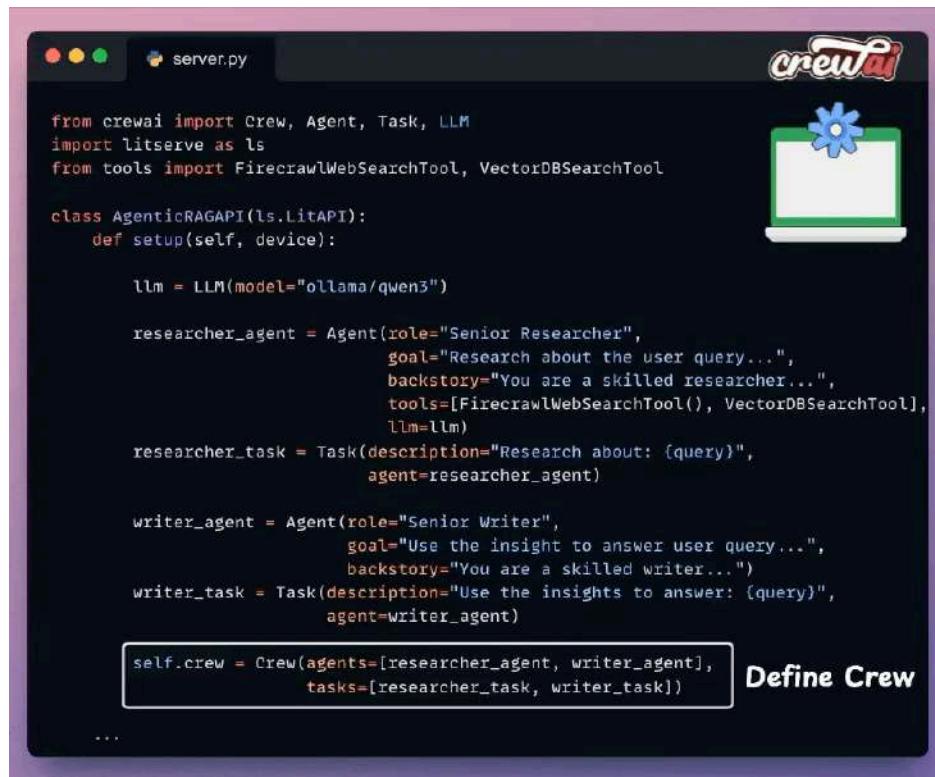
        writer_agent = Agent(role="Senior Writer",
                            goal="Use the insight to answer user query...",
                            backstory="You are a skilled writer...")
        writer_task = Task(description="Use the insights to answer: {query}",
                           agent=writer_agent)
```

Writer Agent

### #4) Set up the Crew

Once we have defined the Agents and their tasks, we orchestrate them into a crew using CrewAI and put that into a setup method.

Check this code:



```

from crewai import Crew, Agent, Task, LLM
import litserve as ls
from tools import FirecrawlWebSearchTool, VectorDBSearchTool

class AgenticRAGAPI(ls.LitAPI):
    def setup(self, device):

        llm = LLM(model="ollama/qwen3")

        researcher_agent = Agent(role="Senior Researcher",
                                  goal="Research about the user query...",
                                  backstory="You are a skilled researcher...",
                                  tools=[FirecrawlWebSearchTool(), VectorDBSearchTool],
                                  llm=llm)
        researcher_task = Task(description="Research about: {query}",
                               agent=researcher_agent)

        writer_agent = Agent(role="Senior Writer",
                             goal="Use the insight to answer user query...",
                             backstory="You are a skilled writer...")
        writer_task = Task(description="Use the insights to answer: {query}",
                           agent=writer_agent)

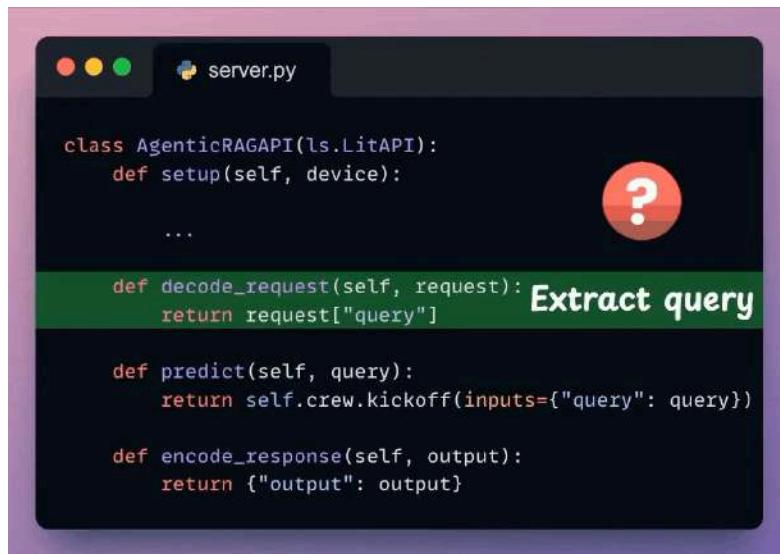
        self.crew = Crew(agents=[researcher_agent, writer_agent],
                        tasks=[researcher_task, writer_task])

```

**Define Crew**

## #5) Decode request

With that, we have orchestrated the Agentic RAG workflow, which will be executed upon an incoming request. Next, from the incoming request body, we extract the user query. Check the highlighted code below:



```

class AgenticRAGAPI(ls.LitAPI):
    def setup(self, device):
        ...

    def decode_request(self, request): Extract query
        return request["query"]

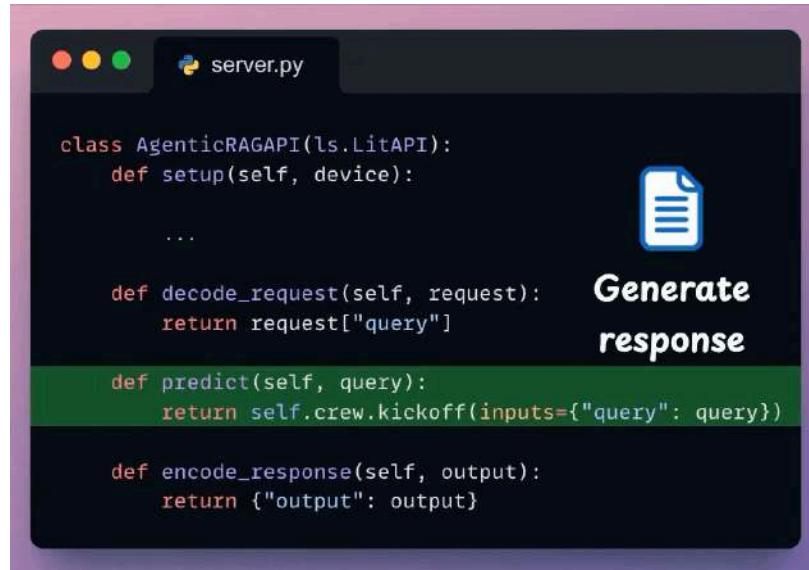
    def predict(self, query):
        return self.crew.kickoff(inputs={"query": query})

    def encode_response(self, output):
        return {"output": output}

```

## #6) Predict

We use the decoded user query and pass it to the Crew defined earlier to generate a response from the model. Check the highlighted code below:



A screenshot of a terminal window titled "server.py". The code is as follows:

```
class AgenticRAGAPI(ls.LitAPI):
    def setup(self, device):
        ...

    def decode_request(self, request):
        return request["query"]

    def predict(self, query):
        return self.crew.kickoff(inputs={"query": query})

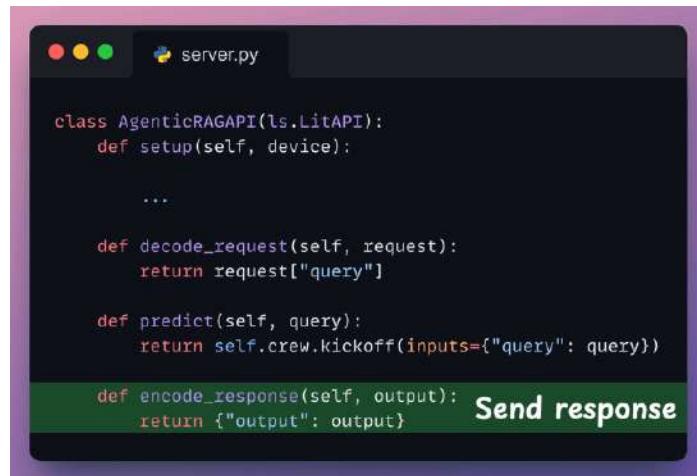
    def encode_response(self, output):
        return {"output": output}
```

The line `b>return self.crew.kickoff(inputs={"query": query})` is highlighted in green and has a blue "Generate response" button next to it.

## #7) Encode response

Here, we can post-process the response & send it back to the client.

Note: LitServe internally invokes these methods in order: decode\_request → predict → encode\_response. Check the highlighted code below:



A screenshot of a terminal window titled "server.py". The code is as follows:

```
class AgenticRAGAPI(ls.LitAPI):
    def setup(self, device):
        ...

    def decode_request(self, request):
        return request["query"]

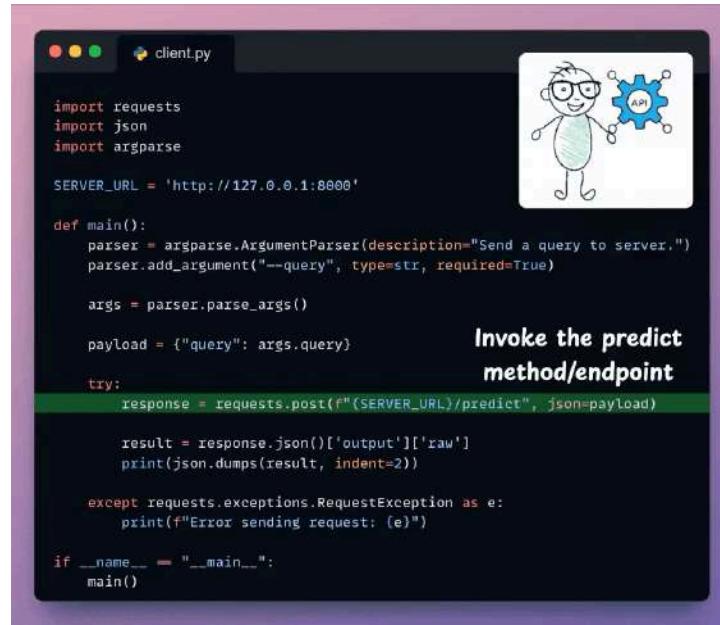
    def predict(self, query):
        return self.crew.kickoff(inputs={"query": query})

    def encode_response(self, output):
        return {"output": output} Send response
```

The line `b>return {"output": output}` is highlighted in green and has a blue "Send response" button next to it.

#8) With that, we are done with the server code.

Next, we have the basic client code to invoke the API we created using the requests Python library. Check this:



```
client.py

import requests
import json
import argparse

SERVER_URL = 'http://127.0.0.1:8000'

def main():
    parser = argparse.ArgumentParser(description="Send a query to server.")
    parser.add_argument("--query", type=str, required=True)

    args = parser.parse_args()

    payload = {"query": args.query}

    try:
        response = requests.post(f"{SERVER_URL}/predict", json=payload)

        result = response.json()['output']['raw']
        print(json.dumps(result, indent=2))

    except requests.exceptions.RequestException as e:
        print(f"Error sending request: {e}")

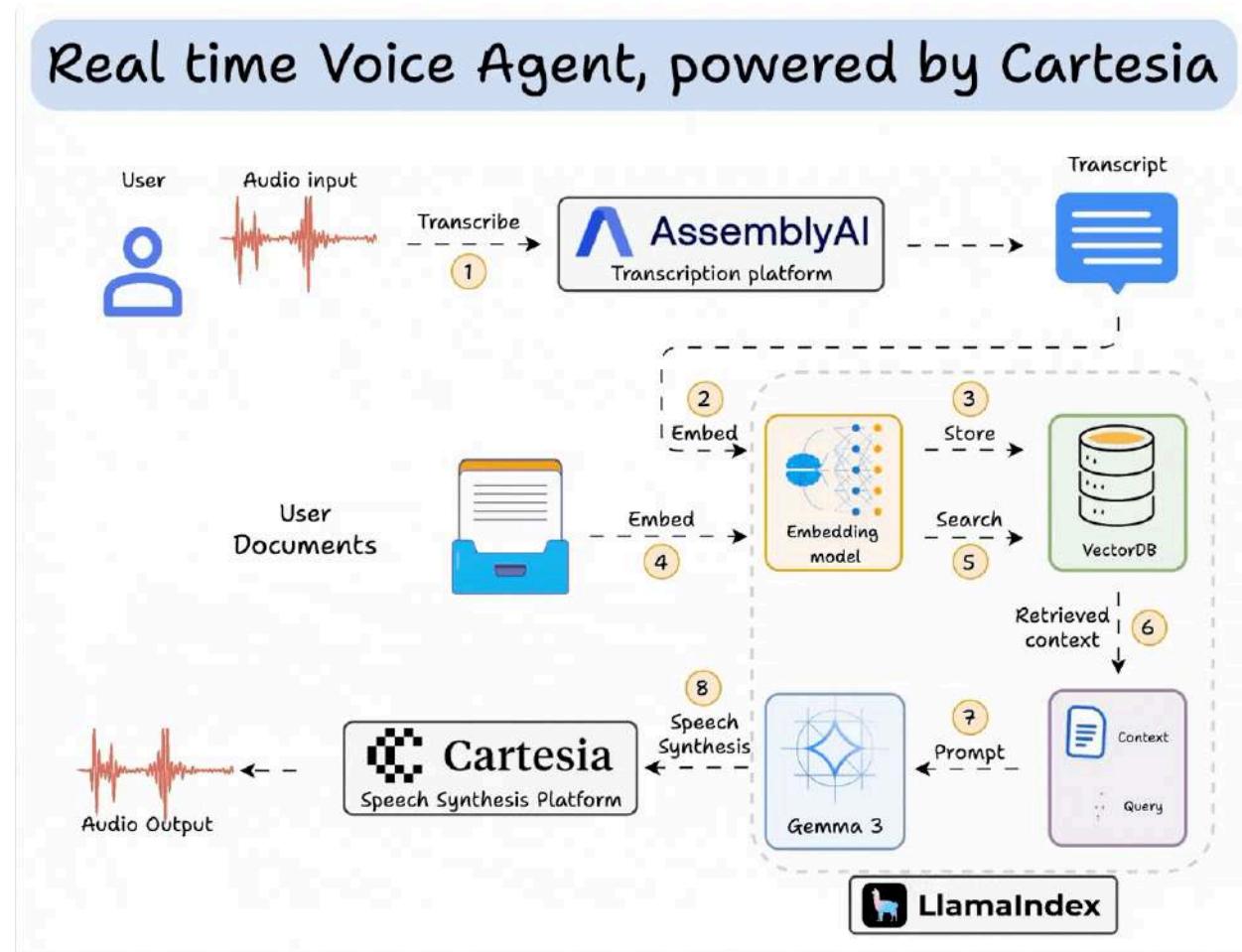
if __name__ == "__main__":
    main()
```

Invoke the predict method/endpoint

Done! We have deployed our fully private Qwen 3 Agentic RAG using LitServe.

## #2) Voice RAG Agent

Real-time voice interactions are becoming more and more popular in AI apps.  
Learn how to build a real-time Voice RAG Agent, step-by-step.



Tech stack:

- CartesiaAI for SOTA text-to-speech
- AssemblyAI for speech-to-text
- LlamaIndex to power RAG
- Livekit for orchestration

Workflow:

- Listens to real-time audio
- Transcribes it via AssemblyAI
- Uses your docs (via LlamaIndex) to craft an answer
- Speaks that answer back with Cartesia

Let's implement this!

### #1) Set up environment and logging

This ensures we can load configurations from .env and keep track of everything in real time.

Check this out:

```
# Environment & Logging Setup

import logging
import os
from dotenv import load_dotenv

# Load environment variables
load_dotenv() ←

# Create a logger for our Voice Assistant
logger = logging.getLogger("voice-assistant")
logger.setLevel(logging.INFO)

# Directory for persisting our index
PERSIST_DIR = "./chat-engine-storage"

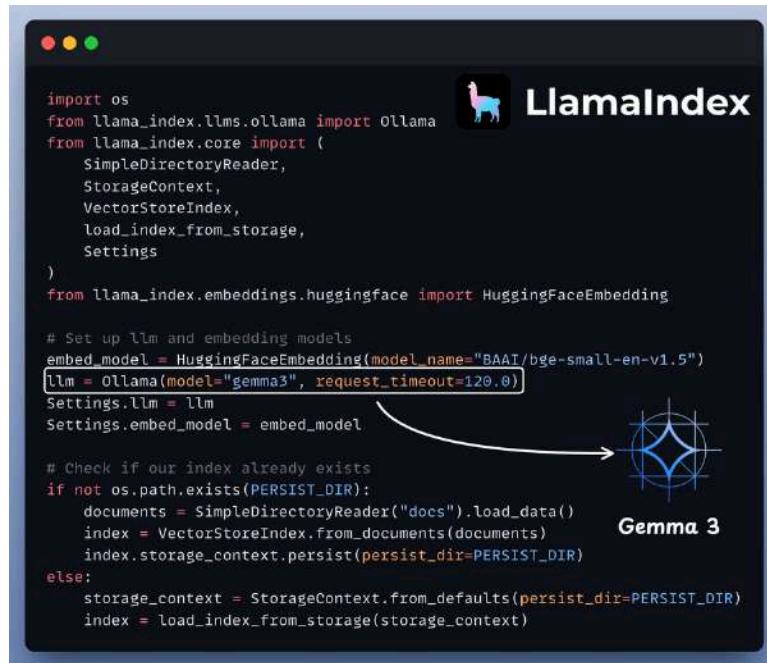

```

The screenshot shows a terminal window with Python code for environment and logging setup. A callout arrow points from the line `load_dotenv()` in the code to a separate window titled ".env" which contains environment variable definitions:

```
CARTESIA_API_KEY=your_cartesia_api_key
LIVEKIT_URL=your_livekit_url
LIVEKIT_API_KEY=your_livekit_api_key
LIVEKIT_API_SECRET=your_livekit_api_secret
ASSEMBLYAI_API_KEY=your_assemblyai_api_key
```

### #2) Setup RAG

This is where your documents get indexed for search and retrieval, powered by LlamaIndex. The agent's answers would be grounded to this knowledge base.



```

import os
from llama_index.llms.ollama import Ollama
from llama_index.core import (
    SimpleDirectoryReader,
    StorageContext,
    VectorStoreIndex,
    load_index_from_storage,
    Settings
)
from llama_index.embeddings.huggingface import HuggingFaceEmbedding

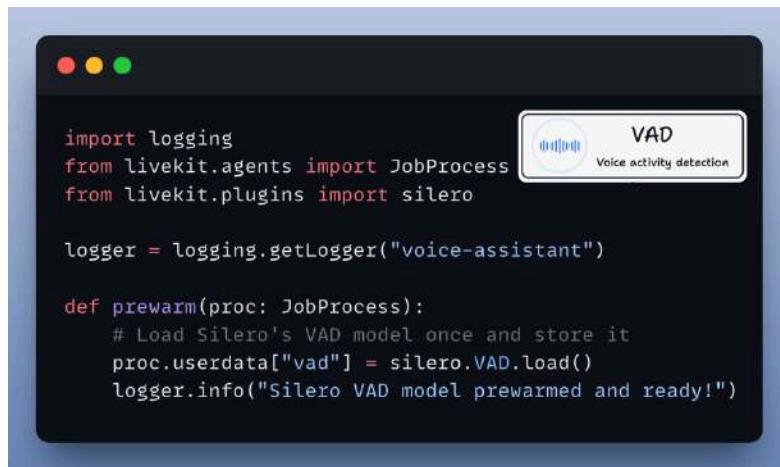
# Set up llm and embedding models
embed_model = HuggingFaceEmbedding(model_name="BAAI/bge-small-en-v1.5")
llm = Ollama(model="gemma3", request_timeout=120.0)
Settings.llm = llm
Settings.embed_model = embed_model

# Check if our index already exists
if not os.path.exists(PERSIST_DIR):
    documents = SimpleDirectoryReader("docs").load_data()
    index = VectorStoreIndex.from_documents(documents)
    index.storage_context.persist(persist_dir=PERSIST_DIR)
else:
    storage_context = StorageContext.from_defaults(persist_dir=PERSIST_DIR)
    index = load_index_from_storage(storage_context)

```

### #3) Setup Voice Activity Detection

We also want Voice Activity Detection (VAD) for smooth real-time experience—so we'll "prewarm" the Silero VAD model. This helps us detect when someone is actually speaking. Check this out:



```

import logging
from livekit.agents import JobProcess
from livekit.plugins import silero

logger = logging.getLogger("voice-assistant")

def prewarm(proc: JobProcess):
    # Load Silero's VAD model once and store it
    proc.userdata["vad"] = silero.VAD.load()
    logger.info("Silero VAD model prewarmed and ready!")

```

### #4) The VoicePipelineAgent and Entry Point

This is where we bring it all together. The agent:

1. Listens to real-time audio.
2. Transcribes it using AssemblyAI.
3. Craft an answer with your documents via LlamaIndex.
4. Speaks that answer back using Cartesia.

Check this out:

```

import logging
from livekit.agents import JobContext
from livekit.agents.pipeline import VoicePipelineAgent

async def entrypoint(ctx: JobContext):
    # Define system role for short, witty replies
    chat_context = ChatContext().append(
        role="system",
        text=(
            "You are a funny, witty assistant, who responds with short, concise answers."
        ),
    )
    # Convert index to a chat engine
    chat_engine = index.as_chat_engine(chat_mode=ChatMode.CONTEXT)
    await ctx.connect(auto_subscribe=AutoSubscribe.AUDIO_ONLY)

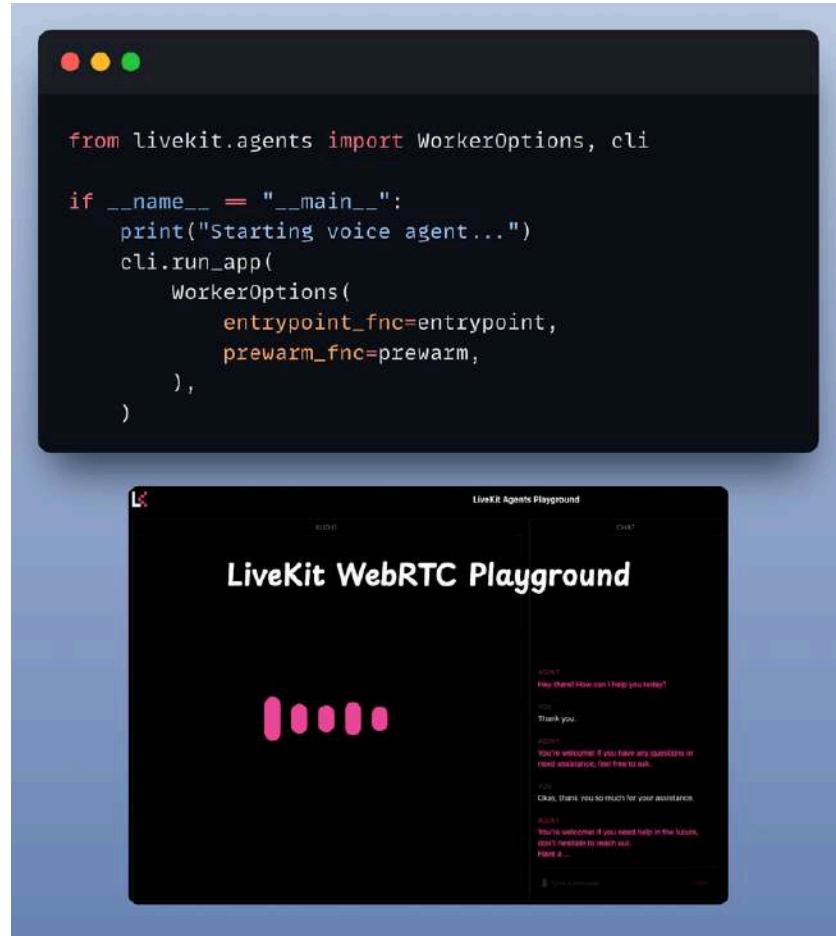
    participant = await ctx.wait_for_participant()
    logger.info(f"Participant {participant.identity} joined! Starting voice assistant.")

    # Build the voice pipeline: VAD → STT → RAG → TTS
    agent = VoicePipelineAgent(
        vad=ctx.proc.userdata["vad"],
        stt=assemblyai.STT(),
        llm=llama_index.LLM(chat_engine=chat_engine),
        tts=cartesia.TTS(
            model="sonic-2",
            voice="794f9389-aac1-45b6-b726-9d9369183238",
        ),
        chat_ctx=chat_context,
    )
    agent.start(ctx.room, participant)
    # Greet the user with a short message
    await agent.say("Hey there! How can I help you today?", allow_interruptions=True)

```

#5) Run the app

Finally, we tie it all together. We run our agent with, specifying the prewarm function and main entrypoint.

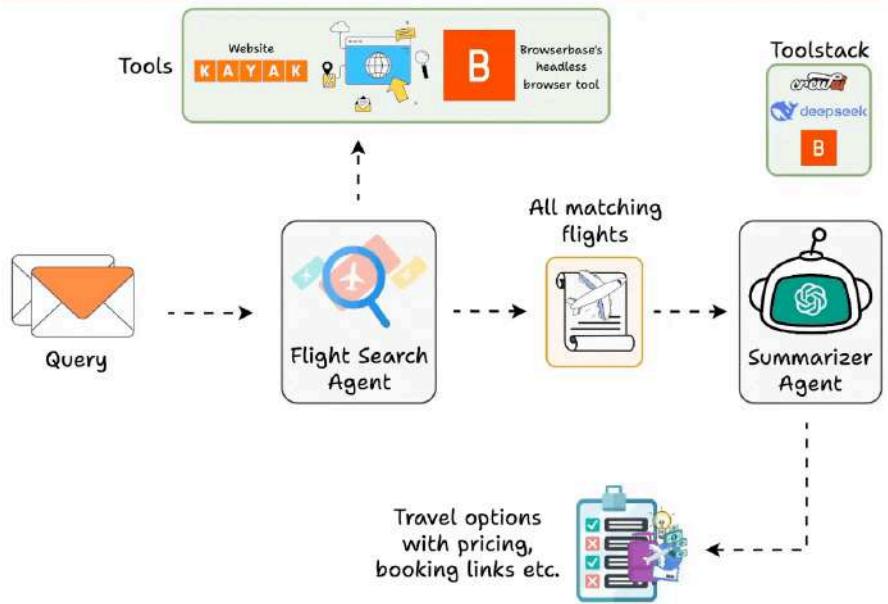


That's it—your Real-Time Voice RAG Agent is ready to roll!

## #3) Multi-agent Flight finder

Build a flight search pipeline with agentic capabilities that can parse natural language queries and fetch live results from Kayak.

### FlightFinder Pro: Find best flights



Tech stack:

- CrewAIInc for multi-agent orchestration
- BrowserbaseHQ's headless browser tool
- Ollama to locally serve DeepSeek-R1

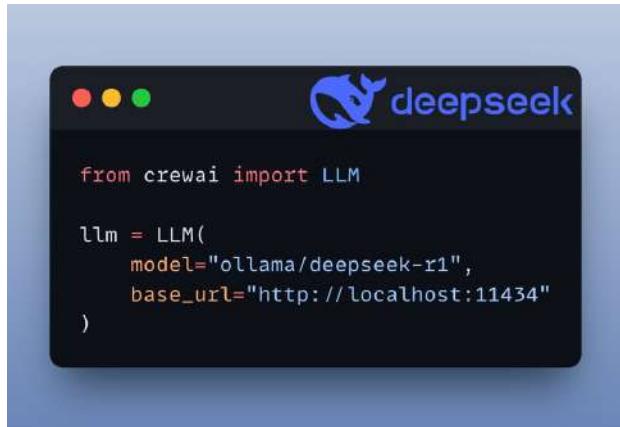
Workflow:

- Parse the query (SF to New York on 21st September) to create a Kayak search URL
- Visit the URL and extract top 5 flights
- For each flight, go to the details to find available airlines
- Summarize flight info

Let's implement this!

## #1) Define LLM

CrewAI nicely integrates with all the popular LLMs and providers out there. Here's how you set up a local DeepSeek using Ollama:

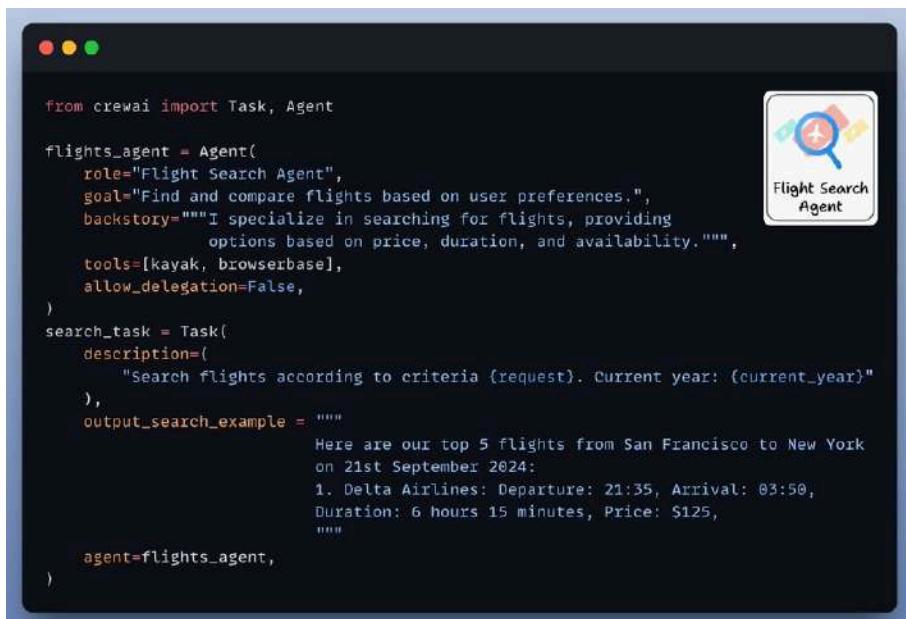


```
from crewai import LLM

llm = LLM(
    model="ollama/deepseek-r1",
    base_url="http://localhost:11434"
)
```

## #2) Flight Search Agent

This agent mimics a real human searching for flights by browsing the web, powered by Browserbase's headless-browser tool and can look up flights on sites like Kayak.



```
from crewai import Task, Agent

flights_agent = Agent(
    role="Flight Search Agent",
    goal="Find and compare flights based on user preferences.",
    backstory="""I specialize in searching for flights, providing
options based on price, duration, and availability.""",
    tools=[kayak, browserbase],
    allow_delegation=False,
)
search_task = Task(
    description=(
        "search flights according to criteria {request}. Current year: {current_year}"
    ),
    output_search_example = """
        Here are our top 5 flights from San Francisco to New York
        on 21st September 2024:
        1. Delta Airlines: Departure: 21:35, Arrival: 03:50,
        Duration: 6 hours 15 minutes, Price: $125,
        """
    agent=flights_agent,
)
```

## #3) Summarisation Agent

After retrieving the flight details, we need a concise summary of all available options.

This is where our Summarization Agent steps in to make sense of the results for easy reading.



```
from crewai import Task, Agent

summarize_agent = Agent(
    role="Summarization Agent",
    goal="Summarise text while preserving key details and clarity.",
    backstory="""I specialize in summarizing content efficiently,
                extracting essential information while maintaining coherence.""",
    allow_delegation=False,
)
summarization_task = Task(
    description="Summarise the raw search results",
    expected_output="""
        Here are our top 5 picks from SF to New York on 21st September 2024:
        1. Delta Airlines:
            - Departure: 21:35
            - Arrival: 03:50
            - Duration: 6 hours 15 minutes
            - Price: $125
            - Booking: [Delta Airlines](https://www.kayak.com/)
            ...
        """
    )
    agent=flights_agent,
)
```

Now that we have both our agents ready, it's time to understand the tools powering them.

1. Kayak tool
2. Browserbase tool

Let's write their code one-by-one.



## #4) Kayak tool

A custom Kayak tool to translate the user input into a valid Kayak search URL.

(FYI Kayak is a popular flight and hotel booking)

```
from crewai.tools import tool
from typing import Optional

@tool("Kayak tool")
def kayak_search(
    departure: str, destination: str, date: str, return_date: Optional[str] = None
) -> str:
    """
    Generates a Kayak URL for flights between departure and destination on the specified date.

    :param departure: The IATA code for the departure airport (e.g., 'SOF' for Sofia)
    :param destination: The IATA code for the destination airport (e.g., 'BER' for Berlin)
    :param date: The date of the flight in the format 'YYYY-MM-DD'
    :param return_date: Only for two-way tickets. The date of return flight in the format 'YYYY-MM-DD'
    :return: The Kayak URL for the flight search
    """

    print(f"Generating Kayak URL for {departure} to {destination} on {date}")
    URL = f"https://www.kayak.com/flights/{departure}-{destination}/{date}"
    if return_date:
        URL += f"/{return_date}"
    URL += "?currency=USD"
    return URL

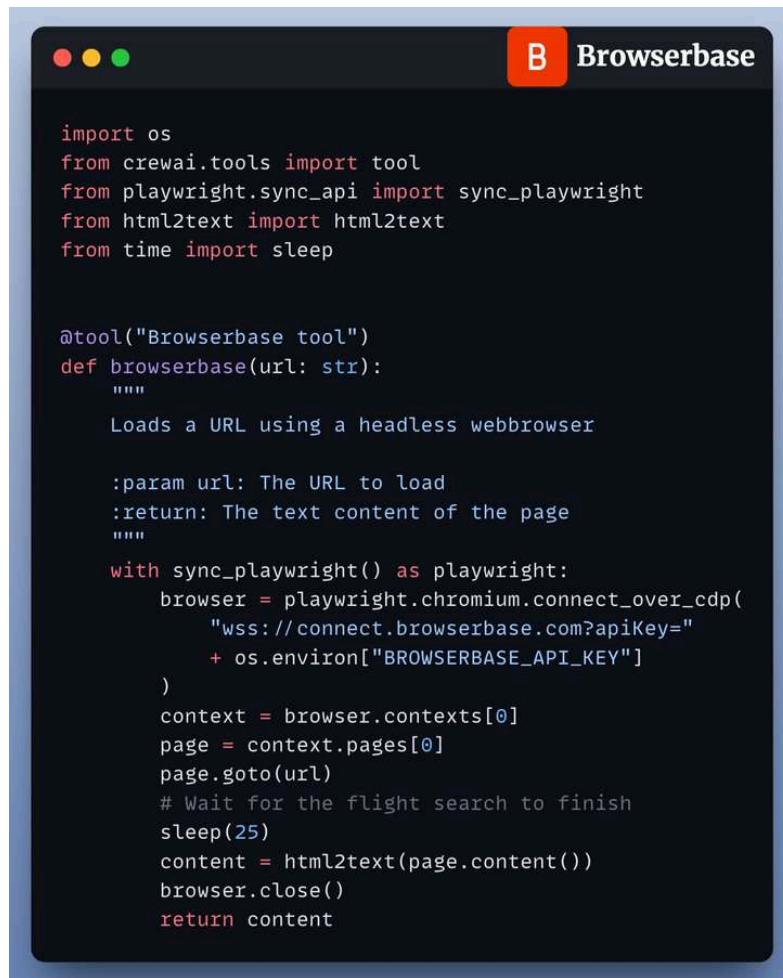
# Export the decorated function
kayak = kayak_search
```

code2career

## #5) Browserbase Tool

The flight search agent uses the Browserbase tool to simulate human browsing and gather flight data.

To be precise it automatically navigates the Kayak website and interacts with the web page. Check this out:



A screenshot of a terminal window titled "Browserbase". The window contains Python code for a tool named "Browserbase tool". The code imports os, crewai.tools, playwright.sync\_api, html2text, and time. It defines a function browserbase(url) that loads a URL using a headless webbrowser. The function takes a parameter url (The URL to load) and returns The text content of the page. Inside the function, it uses sync\_playwright() to connect to a headless browser via CDP. It then navigates to the specified URL, waits for 25 seconds, extracts the page content using html2text, and closes the browser. The code is annotated with docstrings explaining its purpose and parameters.

```
import os
from crewai.tools import tool
from playwright.sync_api import sync_playwright
from html2text import html2text
from time import sleep

@tool("Browserbase tool")
def browserbase(url: str):
    """
    Loads a URL using a headless webbrowser

    :param url: The URL to load
    :return: The text content of the page
    """
    with sync_playwright() as playwright:
        browser = playwright.chromium.connect_over_cdp(
            "wss://connect.browserbase.com?apiKey="
            + os.environ["BROWSERBASE_API_KEY"]
        )
        context = browser.contexts[0]
        page = context.pages[0]
        page.goto(url)
        # Wait for the flight search to finish
        sleep(25)
        content = html2text(page.content())
        browser.close()
    return content
```

## #6) Setup Crew

Once the agents and tools are defined, we orchestrate them using CrewAI.

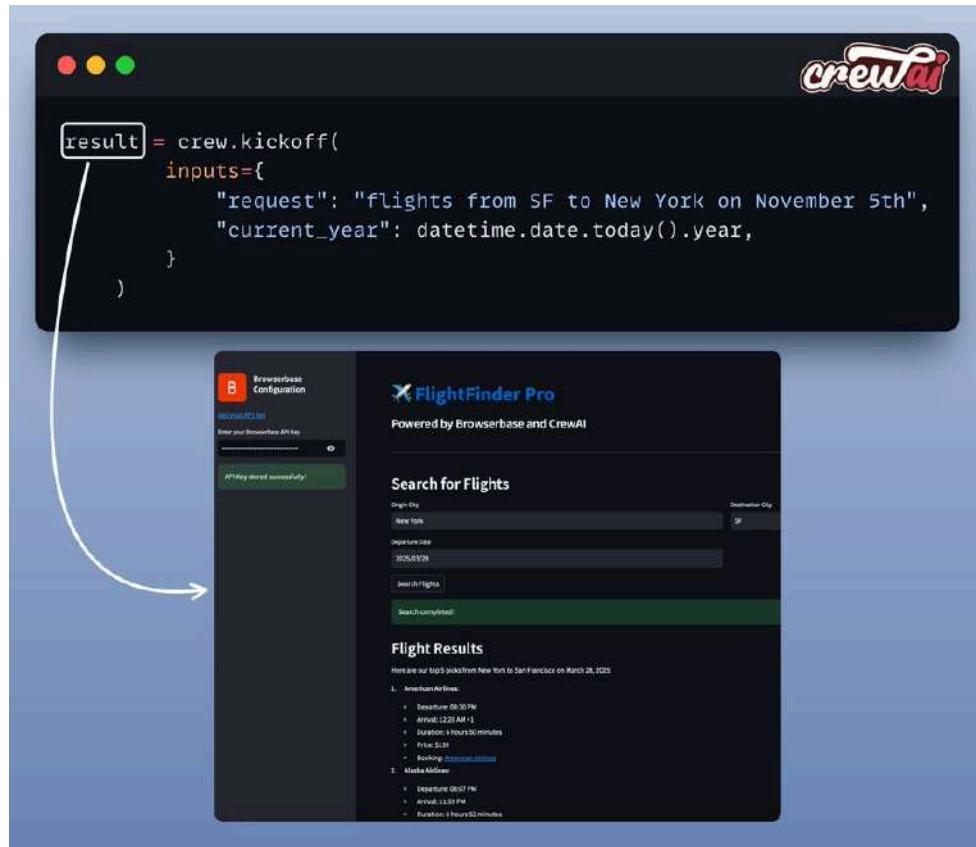
Define their tasks, sequence their actions, and watch them collaborate in real time. Check this out:



*Note: here the 'planning = True' is using the planning pattern we discussed in the 5 Agentic AI Design Patterns above.*

## #7) Kickoff and results

Finally, we feed the user's request (departure city, arrival city, travel dates) into the Crew and let it run:

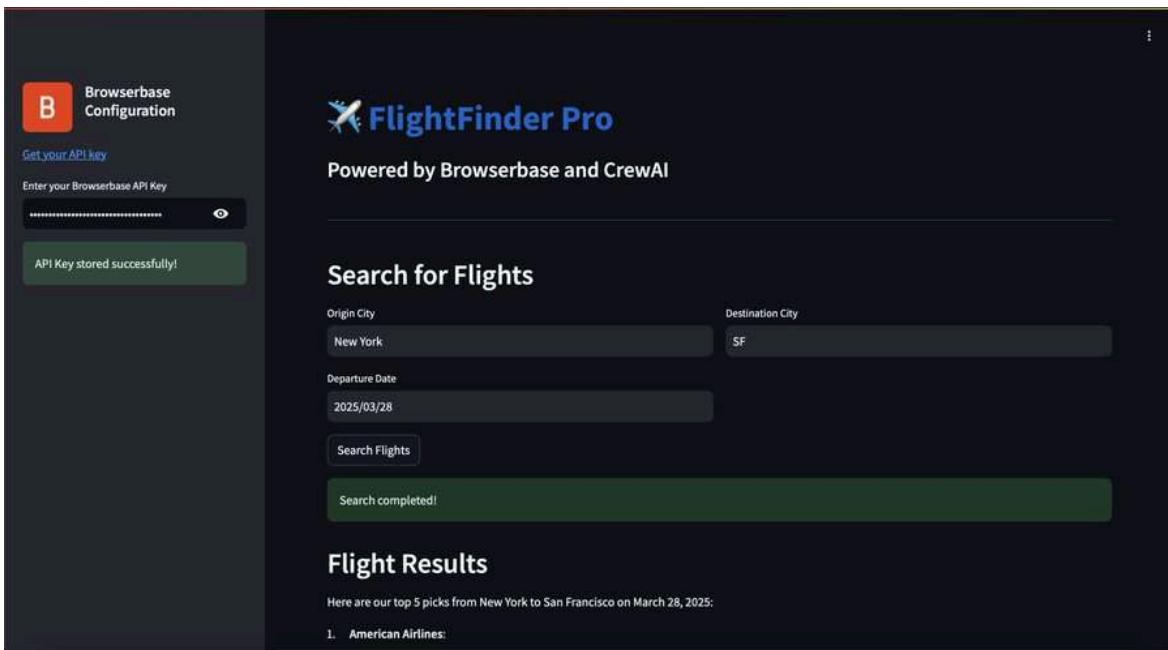


## Streamlit UI

To make this accessible, we wrapped the entire system in a Streamlit interface.

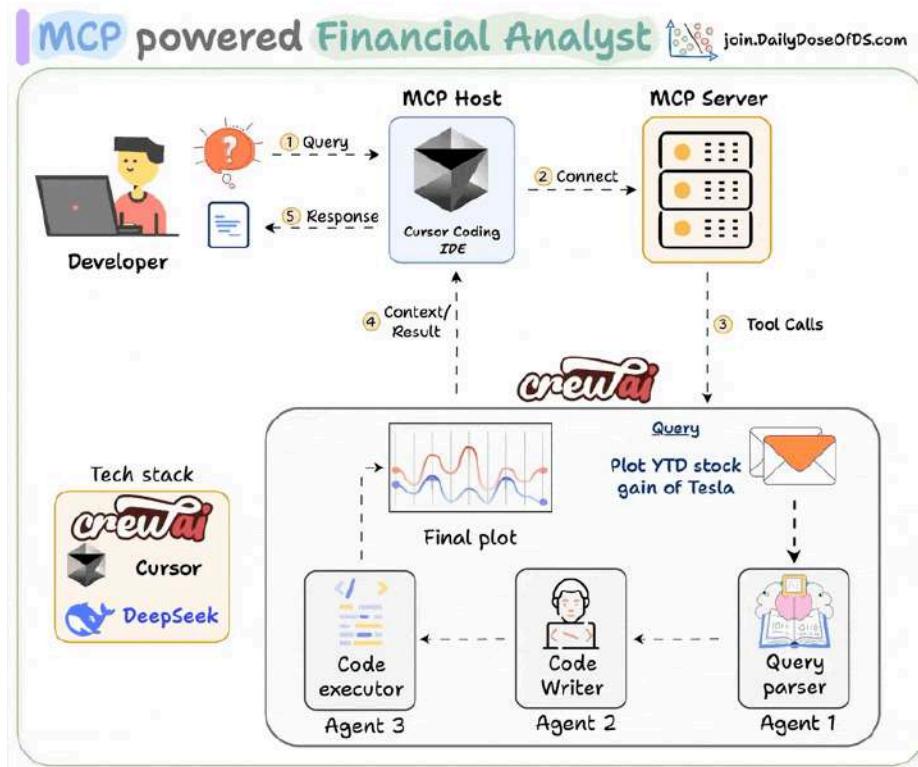
It's a simple chat-like UI where you enter your flight details and see the results in real time.

Check this out:



## #4) Financial Analyst

Build an AI agent that fetches, analyzes & generates insights on stock market trends, right from Cursor or Claude Desktop.



Tech stack:

- CrewAI for multi-agent orchestration
- Ollama to locally serve DeepSeek-R1 LLM
- Cursor as the MCP host

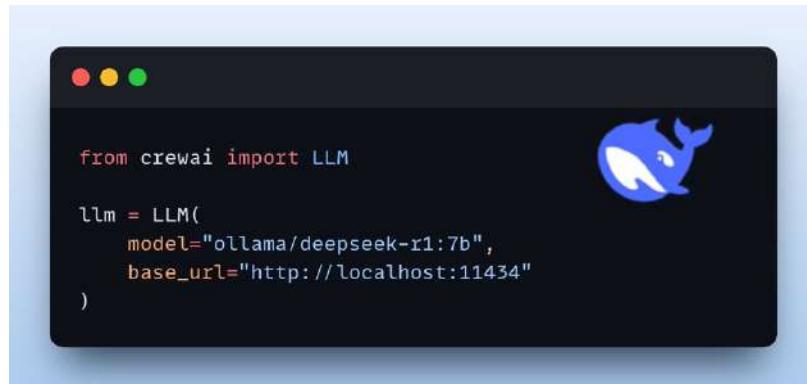
Workflow:

- The user submits a query.
- The MCP agent kicks off the financial analyst crew.
- The crew conducts research and creates an executable script.
- The agent runs the script to generate an analysis plot.

Let's implement this!

## #1) Setup LLM

We will use Deepseek-R1 as the LLM, served locally using Ollama.



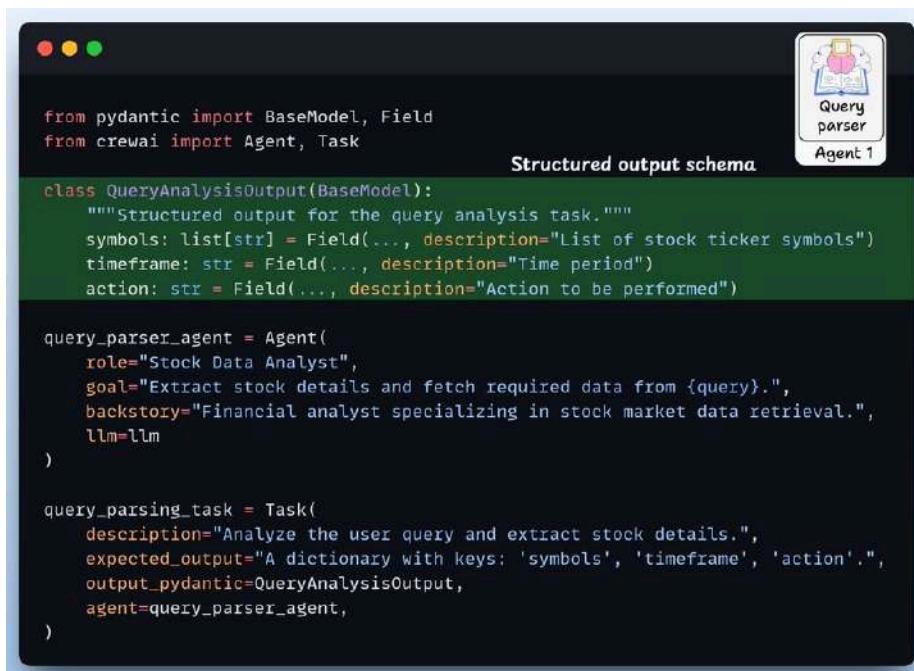
```
from crewai import LLM

llm = LLM(
    model="ollama/deepseek-r1:7b",
    base_url="http://localhost:11434"
)
```

Let's setup the Crew now

## #2) Query Parser Agent

This agent accepts a natural language query and extracts structured output using Pydantic. This guarantees clean and structured inputs for further processing!



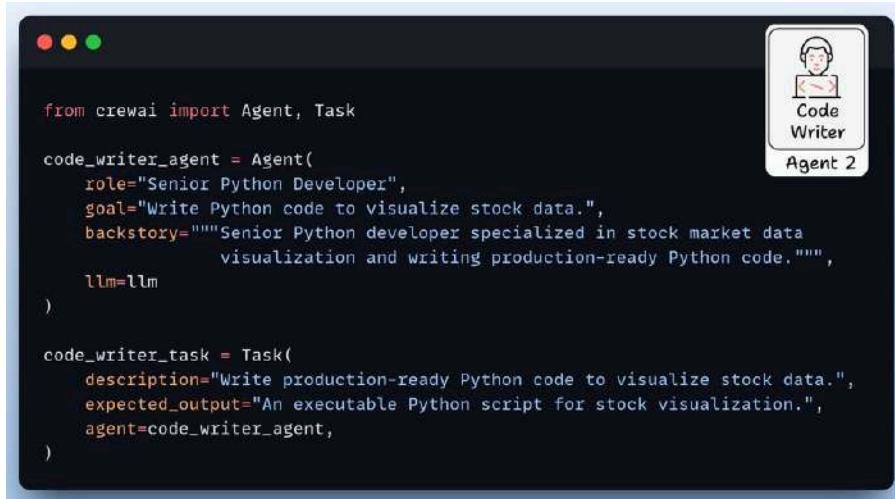
```
from pydantic import BaseModel, Field
from crewai import Agent, Task
from pydantic import BaseModel, Field
from crewai import Agent, Task
Structured output schema
class QueryAnalysisOutput(BaseModel):
    """Structured output for the query analysis task."""
    symbols: list[str] = Field(..., description="List of stock ticker symbols")
    timeframe: str = Field(..., description="Time period")
    action: str = Field(..., description="Action to be performed")

query_parser_agent = Agent(
    role="Stock Data Analyst",
    goal="Extract stock details and fetch required data from {query}.",
    backstory="Financial analyst specializing in stock market data retrieval.",
    llm=llm
)

query_parsing_task = Task(
    description="Analyze the user query and extract stock details.",
    expected_output="A dictionary with keys: 'symbols', 'timeframe', 'action'.",
    output_pydantic=QueryAnalysisOutput,
    agent=query_parser_agent,
)
```

### #3) Code Writer Agent

This agent writes Python code to visualize stock data using Pandas, Matplotlib, and Yahoo Finance libraries.



A screenshot of a terminal window titled "Agent 2". The window contains Python code defining a "code\_writer\_agent" and a "code\_writer\_task". The "code\_writer\_agent" is an Agent with a role of "Senior Python Developer", a goal of "Write Python code to visualize stock data.", and a backstory of "Senior Python developer specialized in stock market data visualization and writing production-ready Python code.". The "code\_writer\_task" is a Task with a description of "Write production-ready Python code to visualize stock data.", an expected output of "An executable Python script for stock visualization.", and an agent of "code\_writer\_agent".

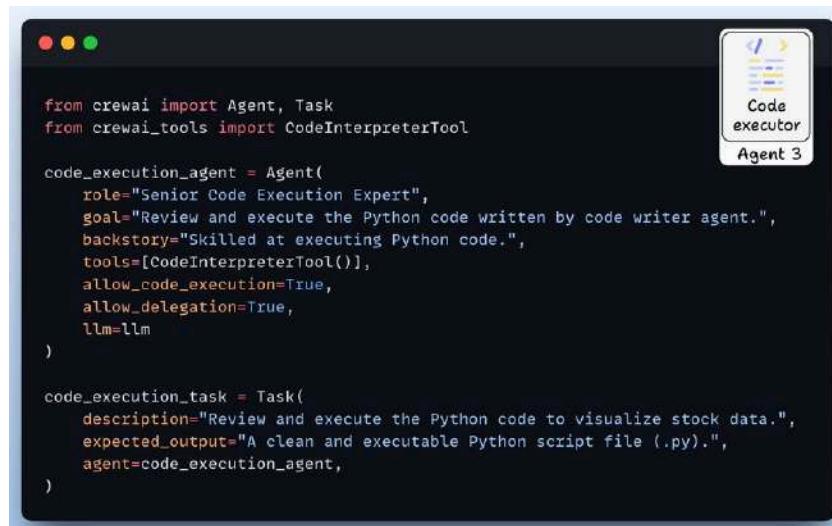
```
from crewai import Agent, Task

code_writer_agent = Agent(
    role="Senior Python Developer",
    goal="Write Python code to visualize stock data.",
    backstory="""Senior Python developer specialized in stock market data
                visualization and writing production-ready Python code."""
)
code_writer_task = Task(
    description="Write production-ready Python code to visualize stock data.",
    expected_output="An executable Python script for stock visualization.",
    agent=code_writer_agent,
)
```

### #4) Code Executor Agent

This agent reviews and executes the generated Python code for stock data visualization.

It uses the code interpreter tool by CrewAI to execute the code in a secure sandbox environment.



A screenshot of a terminal window titled "Agent 3". The window contains Python code defining a "code\_execution\_agent" and a "code\_execution\_task". The "code\_execution\_agent" is an Agent with a role of "Senior Code Execution Expert", a goal of "Review and execute the Python code written by code writer agent.", and a backstory of "Skilled at executing Python code.". The "code\_execution\_task" is a Task with a description of "Review and execute the Python code to visualize stock data.", an expected output of "A clean and executable Python script file (.py).", and an agent of "code\_execution\_agent".

```
from crewai import Agent, Task
from crewai_tools import CodeInterpreterTool

code_execution_agent = Agent(
    role="Senior Code Execution Expert",
    goal="Review and execute the Python code written by code writer agent.",
    backstory="Skilled at executing Python code.",
    tools=[CodeInterpreterTool()],
    allow_code_execution=True,
    allow_delegation=True,
    llm=llm
)

code_execution_task = Task(
    description="Review and execute the Python code to visualize stock data.",
    expected_output="A clean and executable Python script file (.py).",
    agent=code_execution_agent,
)
```

## #5) Setup Crew and Kickoff

We set up and kick off our financial analysis crew to get the result shown below!



## #6) Create MCP Server

Now, we encapsulate our financial analyst within an MCP tool and add two more tools to enhance the user experience.

- save\_code -> Saves generated code to local directory
- run\_code\_and\_show\_plot -> Executes the code and generates a plot

```
from mcp.server.fastmcp import FastMCP
from finance_crew import run_financial_analysis

mcp = FastMCP("financial-analyst")

@mcp.tool()
def analyze_stock(query: str) -> str:
    """Analyzes stock market data based on query and generates executable Python code for analysis and visualization"""
    result = run_financial_analysis(query)
    return result

@mcp.tool()
def save_code(code: str) -> str:
    """Save the given code to a file stock_analysis.py"""
    with open('stock_analysis.py', 'w') as f:
        f.write(code)
    return "Code saved to stock_analysis.py"

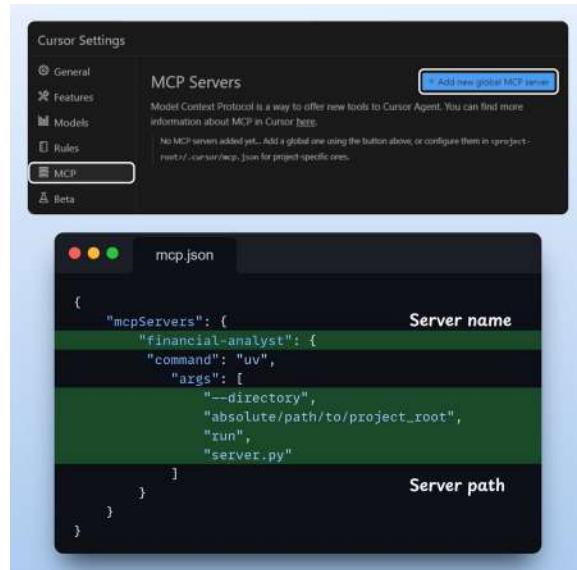
@mcp.tool()
def run_code_and_show_plot() -> str:
    """Execute code and generate a plot"""
    with open('stock_analysis.py', 'r') as f:
        exec(f.read())

if __name__ == "__main__":
    mcp.run(transport='stdio')
```

**Model Context Protocol**

## #7) Integrate MCP server with Cursor

Go to: File → Preferences → Cursor Settings → MCP → Add new global MCP server. In the JSON file, add what's shown below



The screenshot shows the Cursor Settings MCP Servers interface. On the left, there is a sidebar with tabs: General, Features, Models, Rules, MCP (which is selected), and Beta. The main area is titled "MCP Servers" with a sub-instruction: "Model Context Protocol is a way to offer new tools to Cursor Agent. You can find more information about MCP in Cursor here." Below this, it says "No MCP servers added yet... Add a global one using the button above, or configure them in <project>-mcp.json or <cursor>/mcp.json for project-specific ones." A blue-bordered button at the top right says "Add new global MCP server".

Below the interface is a terminal window titled "mcp.json" containing the following JSON code:

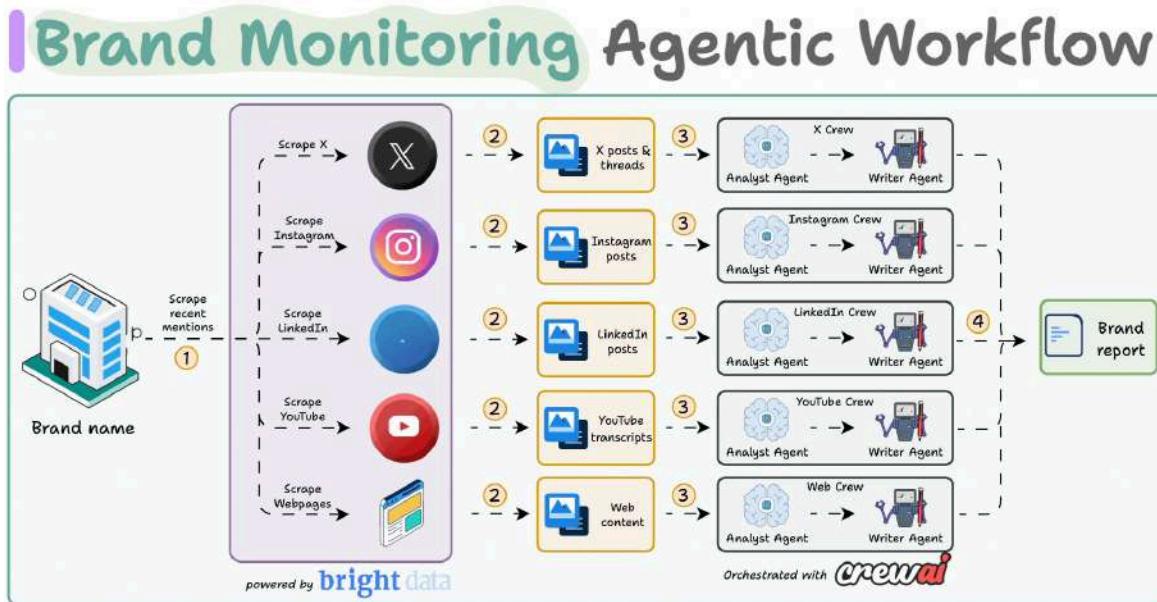
```
{
  "mcpServers": {
    "financial-analyst": {
      "command": "uv",
      "args": [
        "--directory",
        "absolute/path/to/project_root",
        "run",
        "server.py"
      ]
    }
  }
}
```

The code is annotated with labels: "Server name" points to "financial-analyst", and "Server path" points to the "args" section under "command".

Done! Our financial analyst MCP server is live and connected to Cursor.

## #5) Brand Monitoring System

Build a multi agent brand monitoring app that scrapes web mentions and produces insights about a company.



Tech stack:

- Bright Data to scrape data at scale
- CrewAI for orchestration
- Ollama to serve DeepSeek locally

Workflow:

- Use Bright Data to scrape brand mentions across X, Instagram, YouTube, websites, etc.
- Invoke platform-specific Crews to analyze the data and generate insights.
- Merge all insights to get the final report.

Let's implement this!

## #1) Scraping tool

To monitor a brand, we must scrape data across various sources—X, YouTube, Instagram, websites, etc.

Thus, we'll first gather recent search results from Bright Data's SERP API.

See this code:



The screenshot shows a Mac OS X terminal window with a dark theme. The title bar says "book\_writer.py". The window contains Python code for a scraping tool using Bright Data's Web Search Tool. The code defines a class `BrightDataWebSearchTool` that inherits from `BaseTool`. It has an input schema `BrightDataWebSearchToolInput` with a single field `title` of type `str` with a description of "Topic of the book to write about.". The `run` method sends a request to `brd.superproxy.io:33335` using basic authentication with credentials from Bright Data's SERP API. The URL is constructed with the search term `title` and a limit of 500 results. The response is parsed as JSON and the 'organic' results are returned.

```
from crewai.tools import BaseTool
from pydantic import BaseModel, Field
import requests

class BrightDataWebSearchToolInput(BaseModel):
    """Input schema for BrightDataWebSearchTool."""
    title: str = Field(..., description="Topic of the book to write about.")

class BrightDataWebSearchTool(BaseTool):
    name: str = "Web Search Tool"
    description: str = "Tool to search Google and retrieve the results."
    args_schema: Type[BaseModel] = BrightDataWebSearchToolInput

    def _run(self, title):

        host = 'brd.superproxy.io'
        port = 33335

        username = '<Get from brightdata.com in SERP API>'
        password = '<Get from brightdata.com in SERP API>'

        proxies = {
            'http': f'http://{username}:{password}@{host}:{port}',
            'https': f'https://{username}:{password}@{host}:{port}'
        }

        url = f"https://www.google.com/search?q={title}&brd_json=1&num=500"
        response = requests.get(url, proxies=proxies, verify=False)

        return response.json()['organic']
```

## #2) Platform-specific scraping function

The above output will contain links to web pages, X posts, YouTube videos, Instagram posts, etc.

To scrape those sources, we use Bright Data's platform-specific scrapers.

Check this code:



```

def scrape_urls(input_urls: list[str], initial_params: dict, scraping_type: str):
    print(f"Scraping {scraping_type} for {len(input_urls)} urls")

    url = "https://api.brightdata.com/datasets/v3/trigger"

    headers = {
        "Authorization": f"Bearer {os.getenv('BRIGHTDATA_API_KEY')}",
        "Content-Type": "application/json",
    }

    data = [{"url":url} for url in input_urls]

    scraping_response = requests.post(url,
                                       headers=headers,
                                       params=initial_params,
                                       json=data)

    snapshot_id = scraping_response.json()['snapshot_id']

    output_url = f"https://api.brightdata.com/datasets/v3/snapshot/{snapshot_id}"

    output_response = requests.get(output_url,
                                   headers=headers,
                                   params={"format": "json"})
    return output_response.json()

```

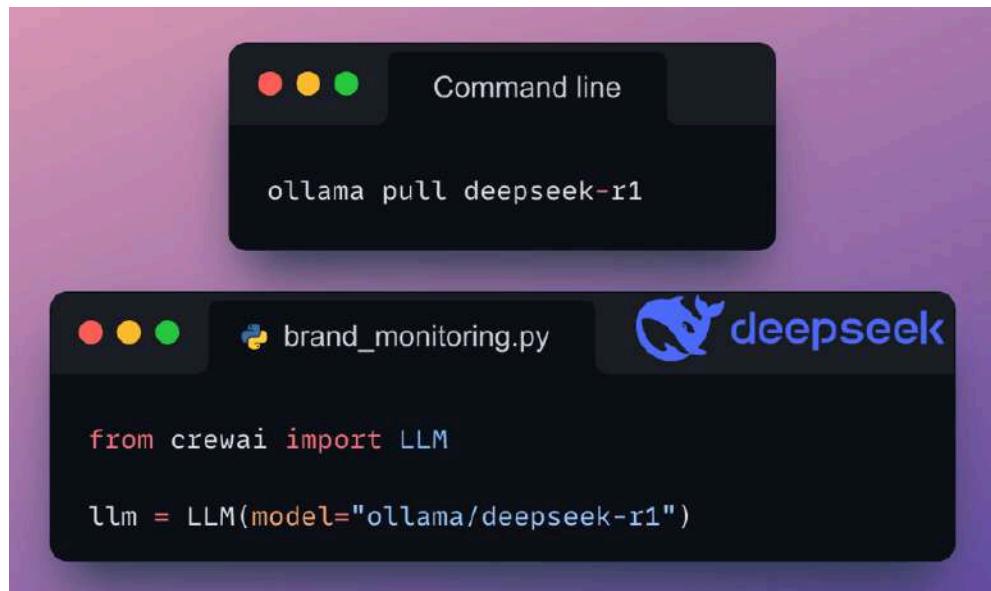
### #3) Set up DeepSeek R1 locally

We'll serve R1 locally through Ollama.

To do this:

- First, we download it locally.
- Next, we define it with the CrewAI's LLM class.

Here's the code:



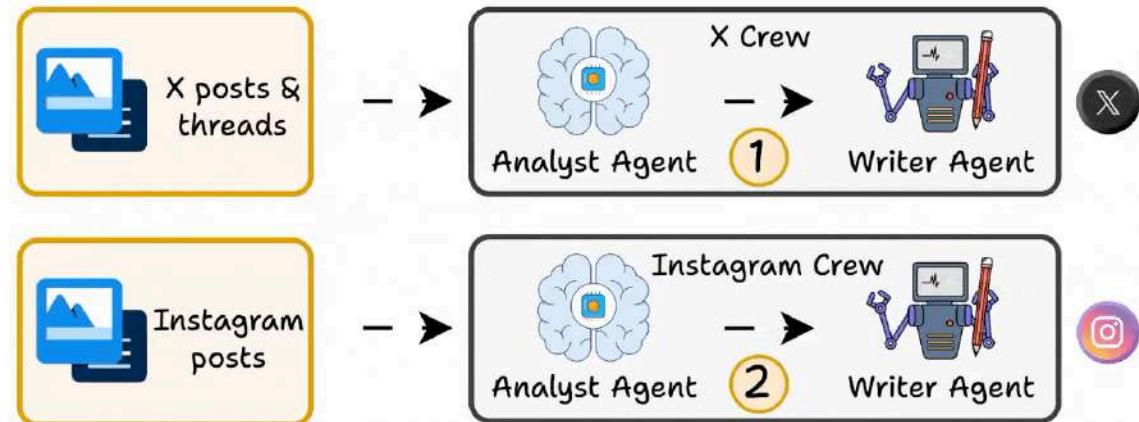
#### #4) Crew Setup

We will have multiple Crews, one for each platform (X, Instagram, YouTube, etc.)

Each Crew will have two Agents:

- Analysis Agent → It analyses the scraped content.
- Writer Agent → It produces insights from the analysis.

Below, let's implement the X Crew!



## #5) X Analyst Agent

This Agent analyzes the posts scraped by Bright Data and extracts key insights. It is also assigned a task to do so.

Here's how it's done:

```

from crewai import Agent, Task

analysis_agent = Agent(role="X Analysis Agent"
    goal="""Analyze the usage of {brand_name} in X posts
        with details like URL, Views, Likes, Replies,
        Reposts, Hashtags, Quotes, Bookmarks, Description,
        Tagged Users, Poster ID"""
    backstory="""You're an X post analysis expert with an
        understanding of the platform and its algorithms.
        You can analyze posts, description, views, likes,
        replies, reposts, hashtags, quotes, bookmarks,
        tagged users and analyse the usage of a brand
        mentioned in the posts."""
)

analysis_task = Task(description="""Analyze the usage of {brand_name} in X posts
        with these details URL, Views, Likes, Replies,
        Reposts, Hashtags, Quotes, Bookmarks, Description,
        Tagged Users, Poster ID.

Scraped X posts This is the list of scraped data:
[x_data]

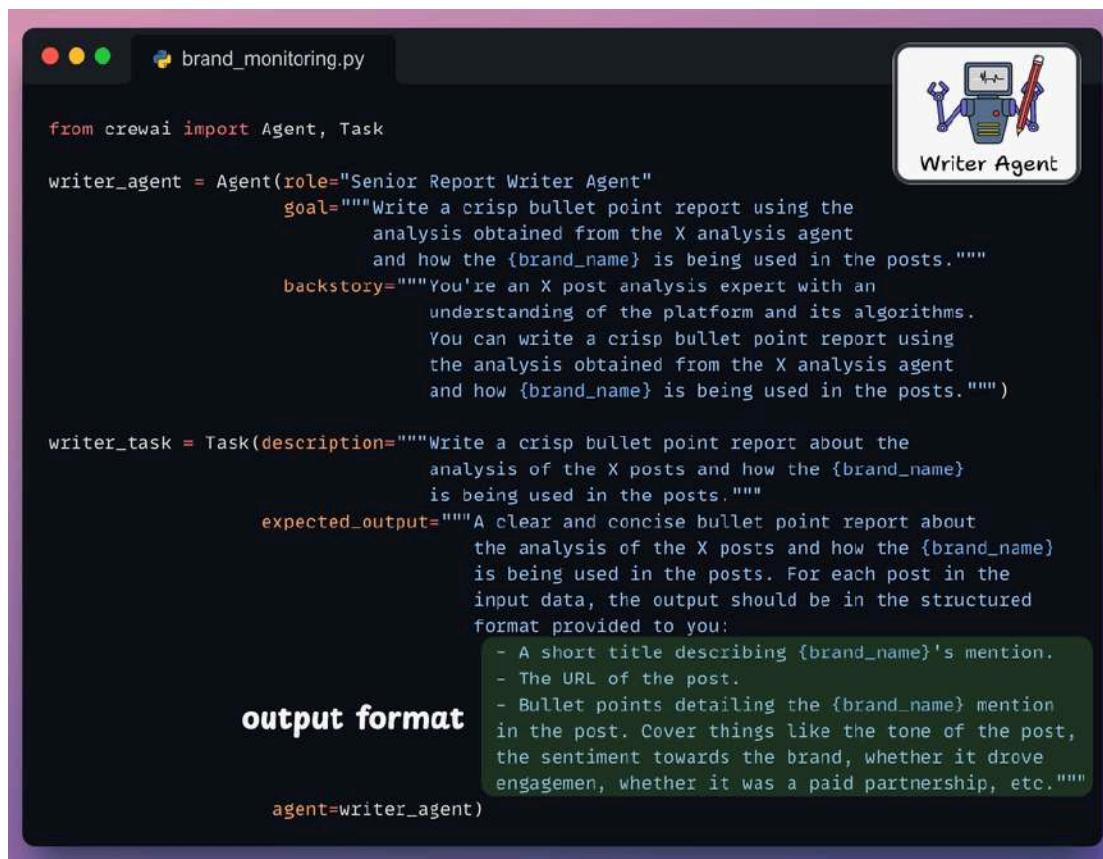
expected_output"""A concise analysis of X posts and how
        {brand_name} is being portrayed in them. You can
        cover things like the tone of the post, the sentiment
        towards the brand, whether it received engagement,
        whether it was a paid partnership, etc."""
    agent=analysis_agent)

```

## #6) X Writer Agent

The Agent takes the output of the X analyst agent and generates insights.

Here's the code:



```

from crewai import Agent, Task

writer_agent = Agent(role="Senior Report Writer Agent"
                     goal="""Write a crisp bullet point report using the
                           analysis obtained from the X analysis agent
                           and how the {brand_name} is being used in the posts."""
                     backstory="""You're an X post analysis expert with an
                           understanding of the platform and its algorithms.
                           You can write a crisp bullet point report using
                           the analysis obtained from the X analysis agent
                           and how {brand_name} is being used in the posts.""")

writer_task = Task(description="""Write a crisp bullet point report about the
                                 analysis of the X posts and how the {brand_name}
                                 is being used in the posts."""
                   expected_output="""A clear and concise bullet point report about
                                     the analysis of the X posts and how the {brand_name}
                                     is being used in the posts. For each post in the
                                     input data, the output should be in the structured
                                     format provided to you:
                                     - A short title describing {brand_name}'s mention.
                                     - The URL of the post.
                                     - Bullet points detailing the {brand_name} mention
                                       in the post. Cover things like the tone of the post,
                                       the sentiment towards the brand, whether it drove
                                       engagement, whether it was a paid partnership, etc.""")

agent=writer_agent)

```

**output format**

## #7) Create a Flow

Finally, we use CrewAI Flows to orchestrate the workflow:

- We start the Flow by using the Scraping tool.
- Next, we invoke platform-specific scrapers.
- Finally, we invoke platform-specific Crews.

Check this out:

```
from crewai.flow import Flow, listen, start
from pydantic import BaseModel

class BrandMonitoringFlow(Flow[BrandMonitoringState]):

    @start()
    def scrape_data(self):
        search_tool = BrightDataWebSearchTool() Scrape URLs of brand mentions
        self.state.search_result = search_tool(self.state.brand_name)

    @listen(scrape_data)
    def scrape_platform_data_and_analyse(self):

        # X scraping
        X_posts = scrape_urls(self.state.search_response.X_urls,
                               X_params, "X")

        X_crew = XCrew().crew()
        X_reponse = X_crew.kickoff(inputs={"X_data": X_posts,
                                           "brand_name": self.state.brand_name}) Scrape X posts

        # YouTube scraping
        YouTube_transcripts = scrape_urls(self.state.search_response.YouTube_urls,
                                            YouTube_params, "X") Scrape YouTube
        YouTube_crew = YouTubeCrew().crew()
        YouTube_reponse = YouTube_crew.kickoff(inputs={"X_data": YouTube_posts,
                                                       "brand_name": self.state.brand_name}) videos

        # repeat for all platforms
```

## #8) Streamlit UI and Kick off the Flow

Finally, we wrap the app in a clear Streamlit interface for interactivity and run the Flow.

Check the final outcome:

The screenshot shows a dark-themed web application interface for "Brand Monitoring powered by deepseek & bright data". On the left, a sidebar titled "Brand Monitoring Settings" contains fields for "Company/Brand Name" (set to "Hugging Face") and "Total Search Results" (set to 25). It also features a red "Start Analysis" button and a "Reset" button. The main content area on the right displays three sections: "LinkedIn Mentions" (with two items), "X/Twitter Mentions" (with three items), and "Web Mentions" (with two items). Each section has a dropdown arrow icon to its right.

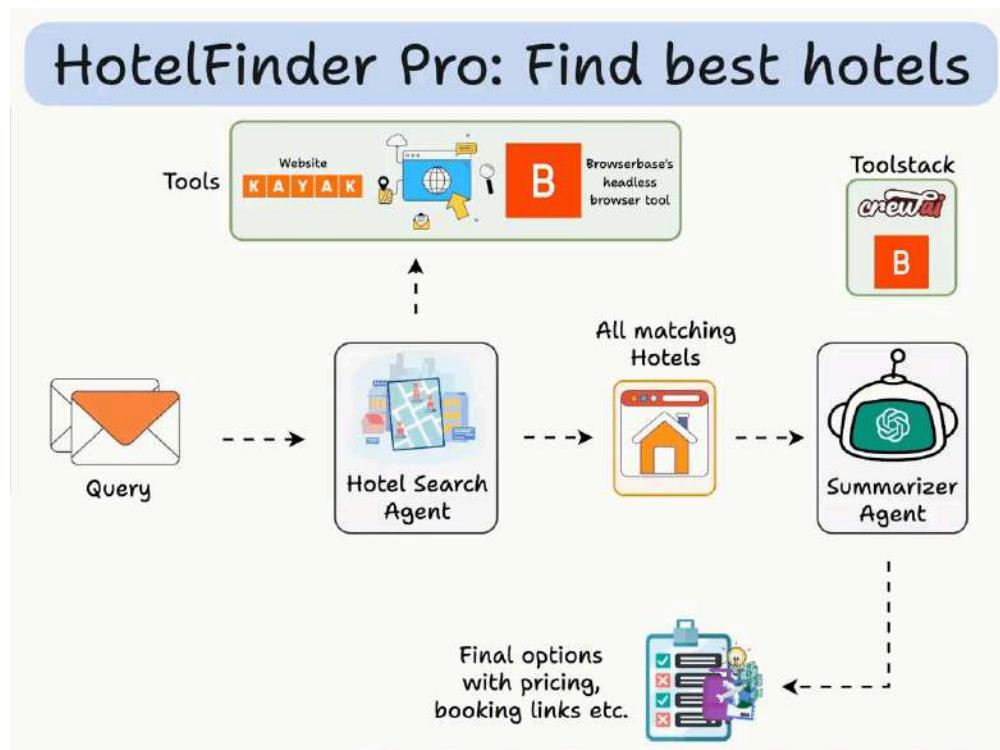
- LinkedIn Mentions**
  - Yann LeCun's Brief Mention of Hugging Face
  - Nazneen Rajani's In-Depth Analysis of AI Risks with Hugging Face
- X/Twitter Mentions**
  - Invitation to Explore the Smartest Model
  - Exciting Partnership Announcement with AlatMeta
  - Joyful Work Culture at Hugging Face
- Web Mentions**
  - Exciting Release of Llama 4 at Hugging Face
  - Technical Guidance for Hugging Face Integrations

Done!

You're now ready to monitor any brand, track all mentions, and generate insights about a company.

## #6) Multi-agent Hotel Finder

Build an Agentic workflow that parses a travel query, fetches live flights and hotel data from Kayak, and summarizes the best options.



Tech stack:

- CrewAI for multi-agent orchestration
- Browserbase's headless browser tool
- Ollama to locally serve DeepSeek-R1

Workflow:

- Parse the query (location, dates etc.) to create a Kayak search URL
- Visit the URL and extract top 5 flights
- For each hotel, find pricing and more info
- Summarize hotel info

Let's implement this!

### #1) Define LLM

CrewAI nicely integrates with all the popular LLMs and providers out there!

Here's how you set up a local DeepSeek using Ollama:



```
from crewai import LLM

llm = LLM(
    model="ollama/deepseek-r1",
    base_url="http://localhost:11434"
)
```

### #2) Hotel Search Agent

This agent mimics a real human searching for hotels by browsing the web. It's powered by browserbase's headless-browser tool and can look up hotels on sites like Kayak.



```
from crewai import Task, Agent

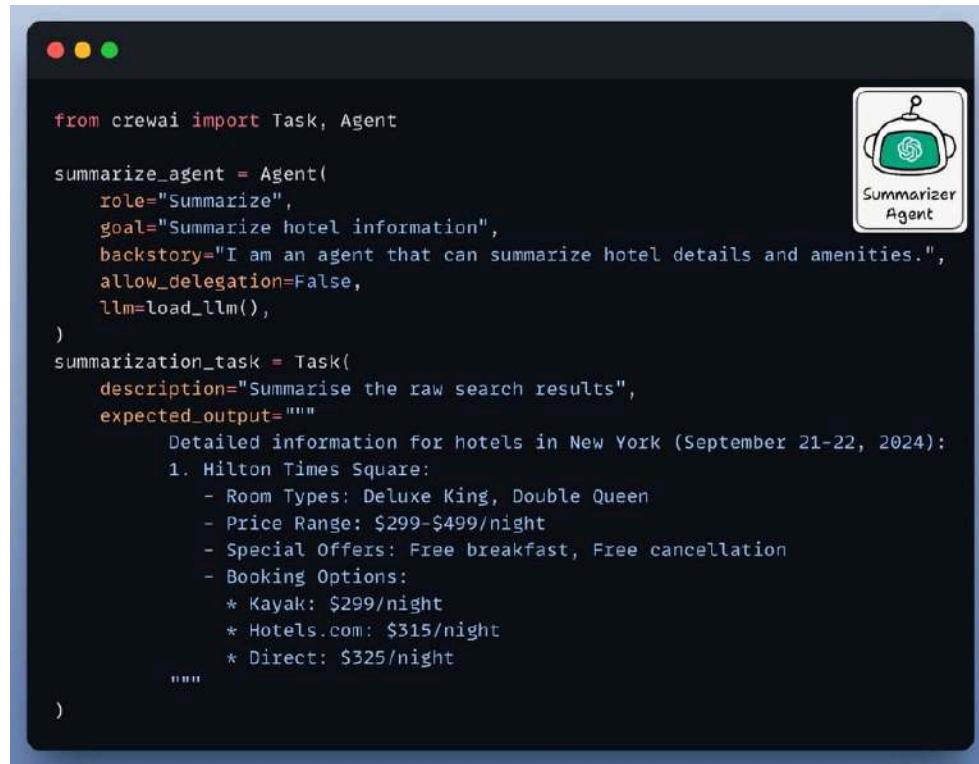
hotels_agent = Agent(
    role="Hotels",
    goal="Search hotels",
    backstory="""I am an agent that can search for hotels and find
        the best accommodations.""",
    tools=[kayak_hotels, browserbase],
    allow_delegation=False,
    llm=load_llm(),
)
search_task = Task(
    description=(
        "Search hotels according to criteria (request). Current year: {current_year}"
    ),
    expected_output="""
        Here are our top 5 hotels in New York for September 21-22, 2024:
        1. Hilton Times Square:
            - Rating: 4.5/5
            - Price: $299/night
            - Location: Times Square
            - Amenities: Pool, Spa, Restaurant
            - Booking: https://www.kayak.com/hotels/hilton-times-square
        """
),
    agent=hotels_agent,
)
```

### #3) Summarisation Agent

After retrieving the hotel details, we need a concise summary of all available options.

This is where our Summarization Agent steps in to make sense of the results for easy reading.

Check this out:



The screenshot shows a terminal window with a dark background. On the right side of the window, there is a small icon of a green teapot with a swirl on it, labeled "Summarizer Agent". The terminal displays the following Python code:

```
from crewai import Task, Agent

summarize_agent = Agent(
    role="Summarize",
    goal="Summarize hotel information",
    backstory="I am an agent that can summarize hotel details and amenities.",
    allow_delegation=False,
    llm=load_llm(),
)
summarization_task = Task(
    description="Summarise the raw search results",
    expected_output="""
        Detailed information for hotels in New York (September 21-22, 2024):
        1. Hilton Times Square:
            - Room Types: Deluxe King, Double Queen
            - Price Range: $299-$499/night
            - Special Offers: Free breakfast, Free cancellation
            - Booking Options:
                * Kayak: $299/night
                * Hotels.com: $315/night
                * Direct: $325/night
        """
)
```

The output of the code is displayed below the code block. It starts with a detailed summary for the Hilton Times Square hotel, listing room types, price ranges, special offers, and booking options.

Now that we have both our agents ready, it's time to understand the tools powering them.

1. Kayak tool

2. Browserbase tool

Let's write their code one-by-one.



## #4) Kayak tool

A custom Kayak tool to translate the user input into a valid Kayak search URL.

(FYI Kayak is a popular for hotel and flight booking)

```

from crewai.tools import tool
from typing import Optional

# --- New Hotel Search Function ---
@tool("Kayak Hotel Tool")
def kayak_hotel_search(
    location: str, check_in_date: str, check_out_date: str, num_adults: int = 2
) -> str:
    """
    Generates a Kayak URL for hotel searches based on location, dates, and number of adults.

    :param location_query: The location string used by Kayak (e.g., 'Hisar,Haryana,India-p15321')
    :param check_in_date: The check-in date in 'YYYY-MM-DD' format
    :param check_out_date: The check-out date in 'YYYY-MM-DD' format
    :param num_adults: The number of adults (defaults to 2)
    :return: The Kayak URL for the hotel search
    """

    URL = f"https://www.kayak.co.in/hotels/{location}/{check_in_date}/{check_out_date}"
    return URL

# Export the new decorated function
kayak_hotels = kayak_hotel_search

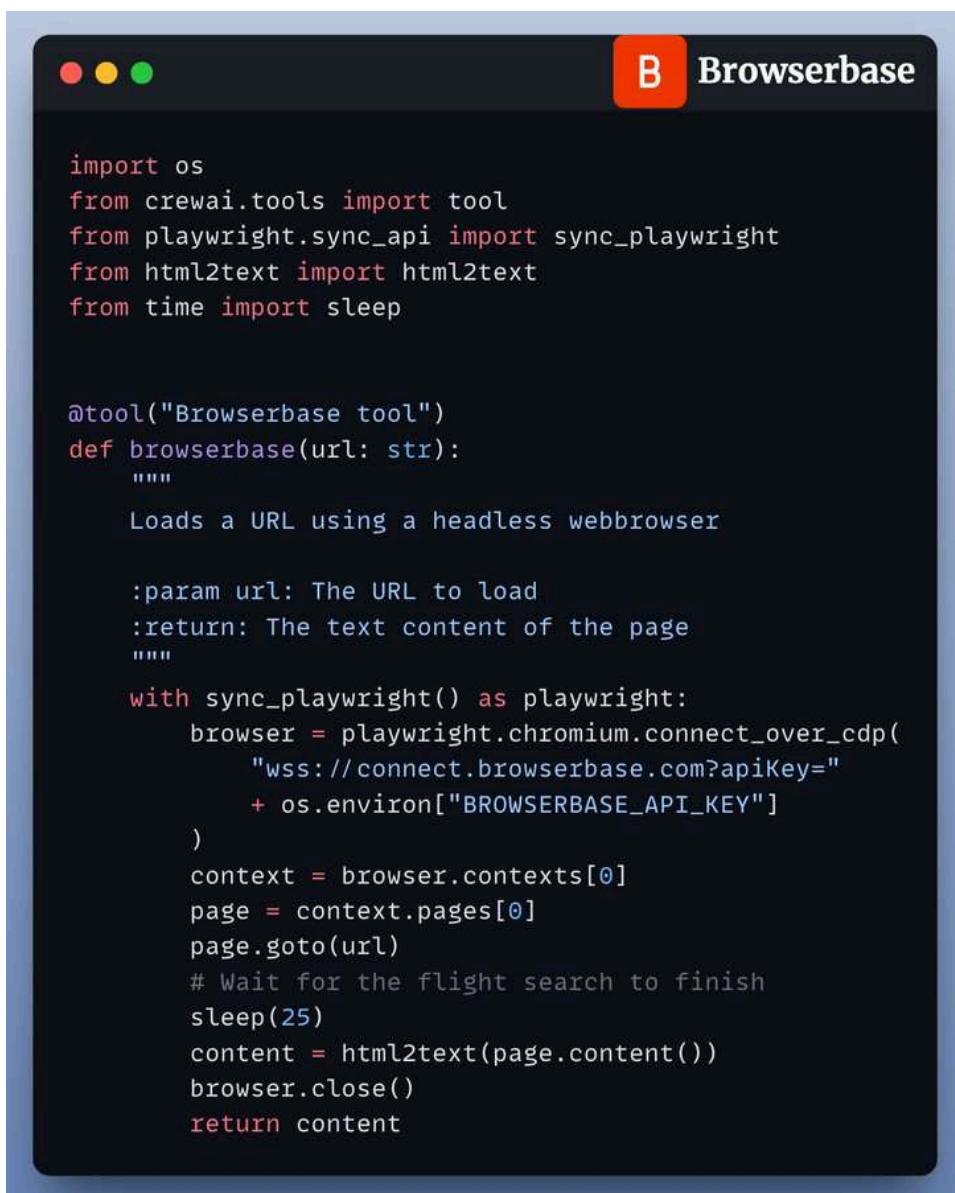
```

## #5) Browserbase Tool

The hotel search agent uses the Browserbase tool to simulate human browsing and gather hotel data.

To be precise it automatically navigates the Kayak website and interacts with the web page.

Check this out:



A screenshot of a terminal window titled "Browserbase". The window contains Python code for a tool named "Browserbase". The code imports os, crewai.tools, playwright.sync\_api, html2text, and time. It defines a function browserbase(url) that loads a URL using a headless webbrowser. The function takes a parameter url (The URL to load) and returns The text content of the page. The code uses playwright to connect over cdp to "wss://connect.browserbase.com?apiKey=" followed by the environment variable "BROWSERBASE\_API\_KEY". It then creates a browser context, opens a new page, goes to the specified URL, waits for 25 seconds, converts the page content to text using html2text, closes the browser, and returns the content.

```
import os
from crewai.tools import tool
from playwright.sync_api import sync_playwright
from html2text import html2text
from time import sleep

@tool("Browserbase tool")
def browserbase(url: str):
    """
    Loads a URL using a headless webbrowser

    :param url: The URL to load
    :return: The text content of the page
    """
    with sync_playwright() as playwright:
        browser = playwright.chromium.connect_over_cdp(
            "wss://connect.browserbase.com?apiKey="
            + os.environ["BROWSERBASE_API_KEY"]
        )
        context = browser.contexts[0]
        page = context.pages[0]
        page.goto(url)
        # Wait for the flight search to finish
        sleep(25)
        content = html2text(page.content())
        browser.close()
    return content
```

## #6) Setup Crew

Once the agents and tools are defined, we orchestrate them using CrewAI.

Define their tasks, sequence their actions, and watch them collaborate in real time! Check this out:

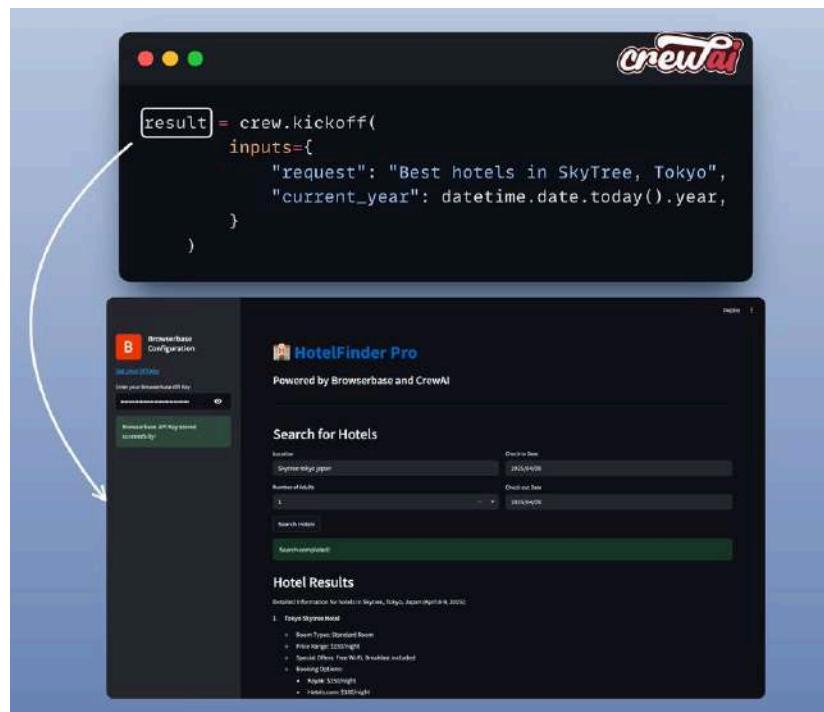


```
from crewai import Crew

crew = Crew(
    agents=[hotels_agent, summarize_agent],
    tasks=[search_task, search_booking_providers_task],
    max_rpm=100,
    verbose=True,
    planning=True,
    llm = load_llm(),
)
```

## #7) Kickoff and results

Finally, we feed the user's request (location, dates etc.) into the Crew and let it run! Check this out:



The image shows two stacked screenshots. The top screenshot is a terminal window with a dark background and white text. It displays the following Python code:

```
result = crew.kickoff(
    inputs={
        "request": "Best hotels in SkyTree, Tokyo",
        "current_year": datetime.date.today().year,
    }
)
```

A white arrow points from the bottom of this terminal window down to a second screenshot below it. The second screenshot is a web browser window titled "HotelFinder Pro". The URL bar shows "https://hotel-finder-pro.com". The page content includes:

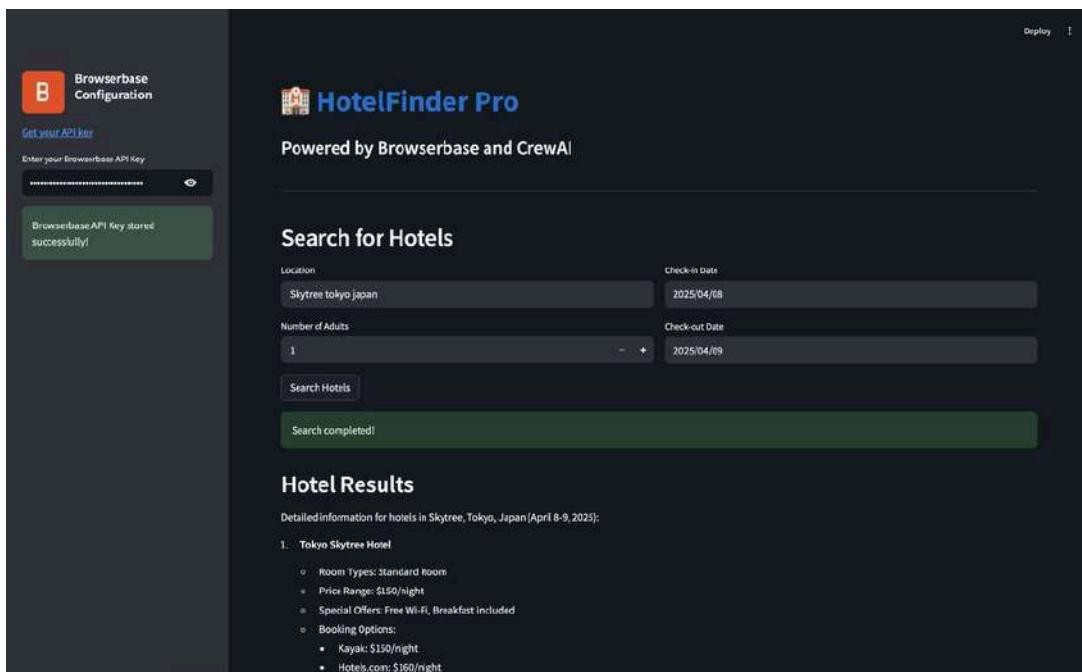
- Browsorbbase Configuration**: A sidebar with a "B" icon and the text "Browsorbbase API Key generated successfully".
- HotelFinder Pro**: Powered by Browsorbbase and CrewAI.
- Search for Hotels**: Form fields for "Location" (Shinjuku, Japan), "Check-in Date" (2023/9/18), "Number of Adults" (1), "Check-out Date" (2023/9/18), and "Search Hotels".
- Hotel Results**: A table with one row for "Tokyo Skytree Hotel". The table includes columns for "Room Type", "Standard Room", "Price per Night", "Total Price for All Nights Booked", and "Booking Options".

## Streamlit UI

To make this accessible, we wrapped the entire system in a @Streamlit interface.

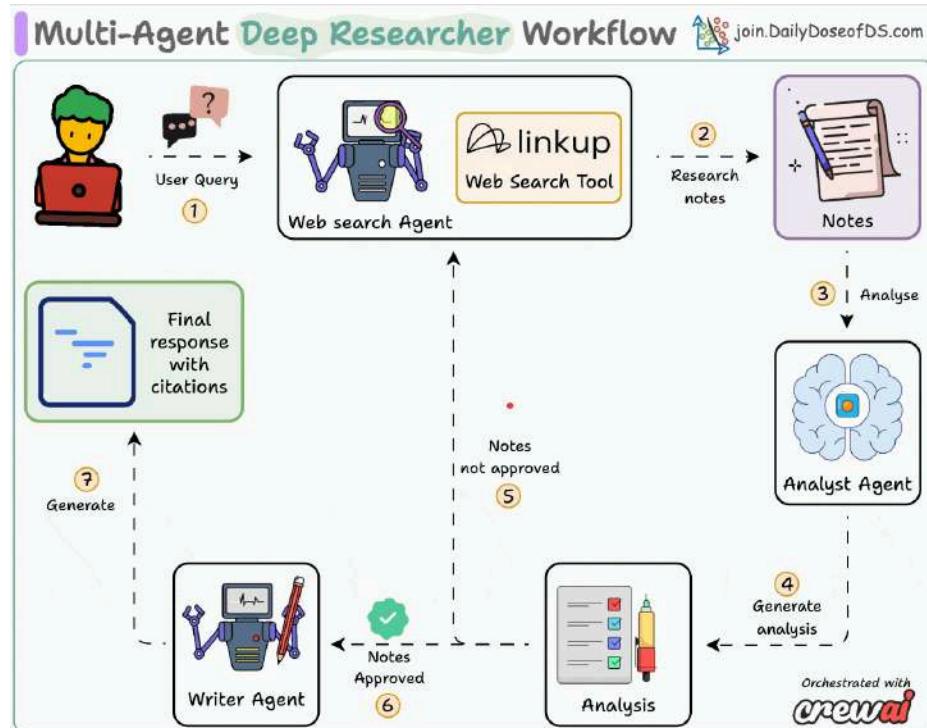
It's a simple chat-like UI where you enter your location and other details and see the results in real time!

Check this out:



## #7) Multi-agent Deep Researcher

ChatGPT has a deep research feature. It helps you get detailed insights on any topic. Learn how you can build a 100% local alternative to it.



Tech stack:

- Linkup platform for deep web research
- CrewAI for multi-agent orchestration
- Ollama to locally serve DeepSeek
- Cursor as MCP host

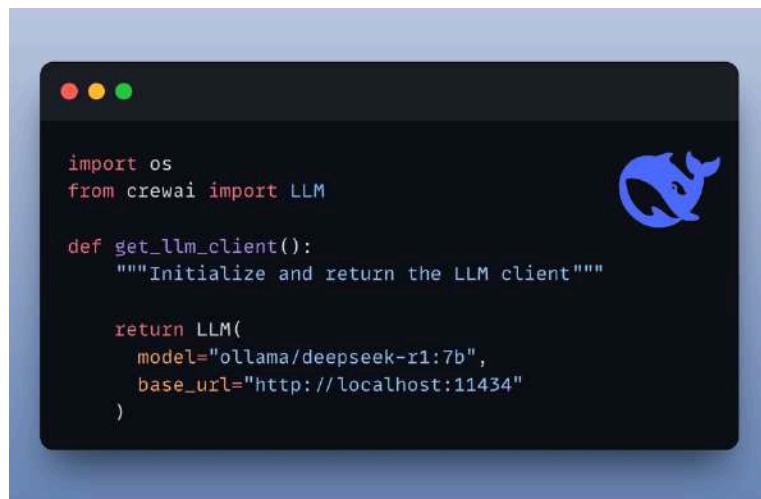
Workflow:

- User submits a query
- Web search agent runs deep web search via Linkup
- Research analyst verifies and deduplicates results
- Technical writer crafts a coherent response with citations

Let's implement this!

### #1) Setup LLM

We'll use a locally served DeepSeek-R1 using Ollama.



```
import os
from crewai import LLM

def get_llm_client():
    """Initialize and return the LLM client"""

    return LLM(
        model="ollama/deepseek-r1:7b",
        base_url="http://localhost:11434"
    )
```

### #2) Define Web Search Tool

We'll use Linkup platform's powerful search capabilities, which rival Perplexity and OpenAI, to power our web search agent. This is done by defining a custom tool that our agent can use.



```

import os
from typing import Type
from pydantic import BaseModel, Field
from linkup import LinkupClient
from crewai.tools import BaseTool

class LinkUpSearchInput(BaseModel):
    """Input schema for Linkup Search Tool."""
    query: str = Field(description="The search query to perform")
    depth: str = Field(default="standard", description="Depth of search: 'standard' or 'deep'")
    output_type: str = Field(
        default="searchResults",
        description="Output type: 'searchResults' or 'sourcedAnswer'"
    )

class LinkUpSearchTool(BaseTool):
    name: str = "LinkUp Search"
    description: str = "Retrieve info from the web using LinkUp and return results"
    args_schema: Type[BaseModel] = LinkUpSearchInput

    def _run(self, query: str, depth: str = "standard",
            output_type: str = "searchResults") -> str:
        """Execute Linkup search and return results."""
        # Initialize Linkup client with API key from environment variables
        linkup_client = LinkupClient(api_key=os.getenv("LINKUP_API_KEY"))

        # Perform search
        search_response = linkup_client.search(query=query,
                                                depth=depth,
                                                output_type=output_type)
        return str(search_response)

```

### #3) Define Web Search Agent

The web search agent gathers up-to-date information from the internet based on user query. The linkup tool we defined earlier is used by this agent.



```

from crewai import Agent, Task

linkup_search_tool = LinkUpSearchTool()

client = get_llm_client()

web_searcher = Agent(
    role="Web Searcher",
    goal="Retrieve relevant info with citations (source URLs)",
    backstory="Expert searcher; forwards results to Research Analyst only.",
    verbose=True,
    allow_delegation=True,
    tools=[linkup_search_tool],
)

search_task = Task(
    description=f"Search for comprehensive information about: {query}.",
    agent=web_searcher,
    expected_output="Detailed raw search results including sources (urls).",
    tools=[linkup_search_tool]
)

```

### #4) Define Research Analyst Agent

This agent transforms raw web search results into structured insights, with source URLs. It can also delegate tasks back to the web search agent for verification and fact-checking.



```
from crewai import Agent, Task

# Define the research analyst
research_analyst = Agent(
    role="Research Analyst",
    goal="Turn raw info into structured insights with URLs.",
    backstory="""Expert analyst; can delegate fact-checks to Web Searcher;
    hands final output to Technical Writer.""",
    verbose=True,
    allow_delegation=True,
    llm=client,
)

analysis_task = Task(
    description="Analyze search results, extract insights, and verify facts.",
    agent=research_analyst,
    expected_output="Structured insights with verified facts and source URLs.",
    context=[search_task],
)
```

## #5) Define Technical Writer Agent

It takes the analyzed and verified results from the analyst agent and drafts a coherent response with citations for the end user.



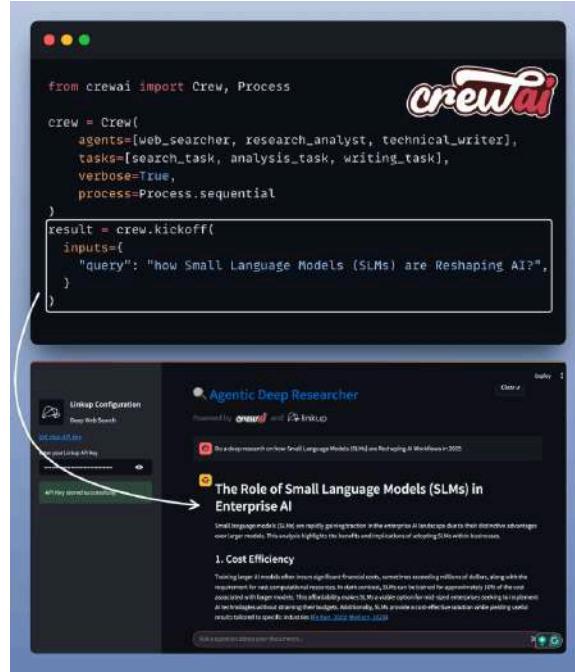
```
from crewai import Agent, Task

technical_writer = Agent(
    role="Technical Writer",
    goal="Create clear markdown responses with citations and source URLs.",
    backstory="Expert in simplifying complex information.",
    verbose=True,
    allow_delegation=False,
)

writing_task = Task(
    description="Write a clear, organized response based on research.",
    agent=technical_writer,
    expected_output="Comprehensive answer with citations and source URLs.",
    context=[analysis_task],
)
```

## #6) Setup Crew

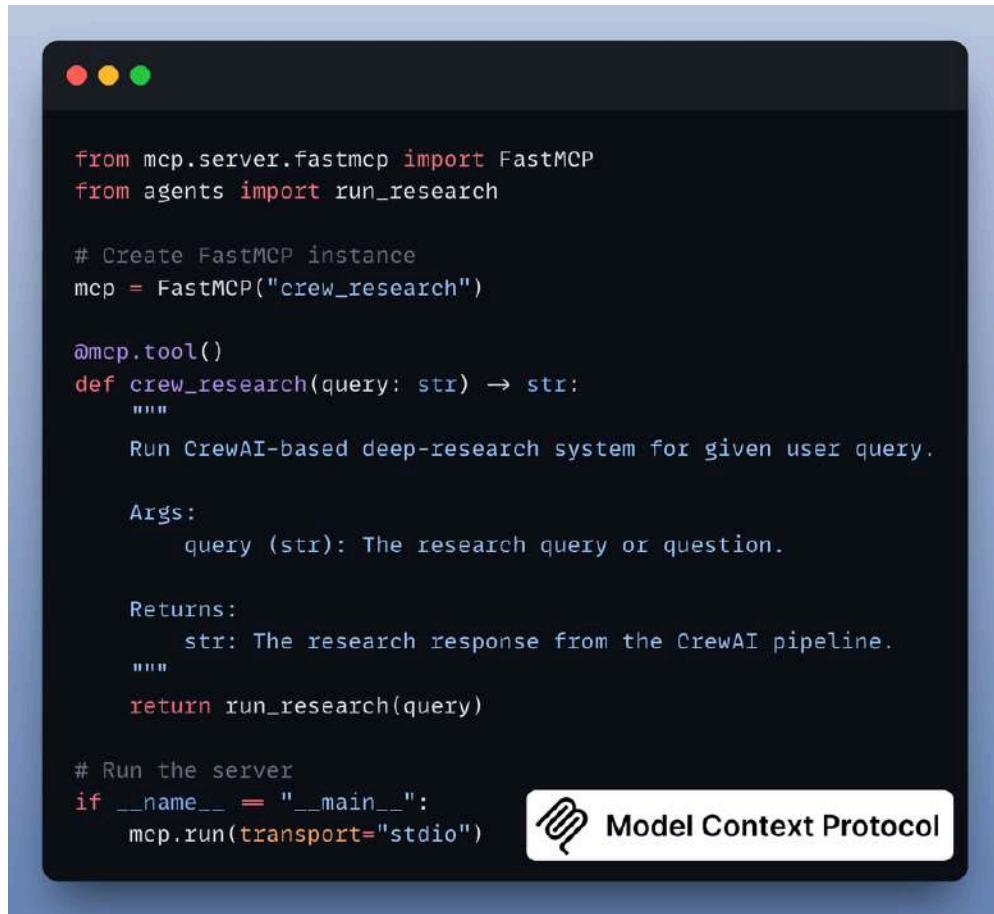
Finally, once we have all the agents and tools defined we set up and kickoff our deep researcher crew.



## #7) Create MCP Server

Now, we'll encapsulate our deep research team within an MCP tool. With just a few lines of code, our MCP server will be ready.

Let's see how to connect it with Cursor.



```
from mcp.server.fastmcp import FastMCP
from agents import run_research

# Create FastMCP instance
mcp = FastMCP("crew_research")

@mcp.tool()
def crew_research(query: str) -> str:
    """
    Run CrewAI-based deep-research system for given user query.

    Args:
        query (str): The research query or question.

    Returns:
        str: The research response from the CrewAI pipeline.
    """
    return run_research(query)

# Run the server
if __name__ == "__main__":
    mcp.run(transport="stdio")
```



Model Context Protocol

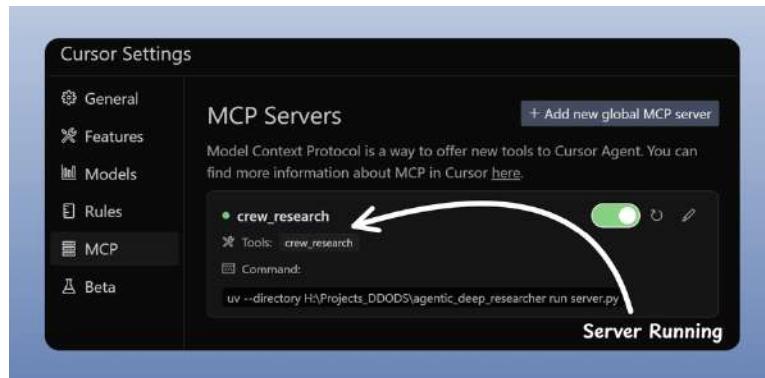
## #8) Integrate MCP server with Cursor

Go to: File → Preferences → Cursor Settings → MCP → Add new global MCP server

In the JSON file, add what's shown below

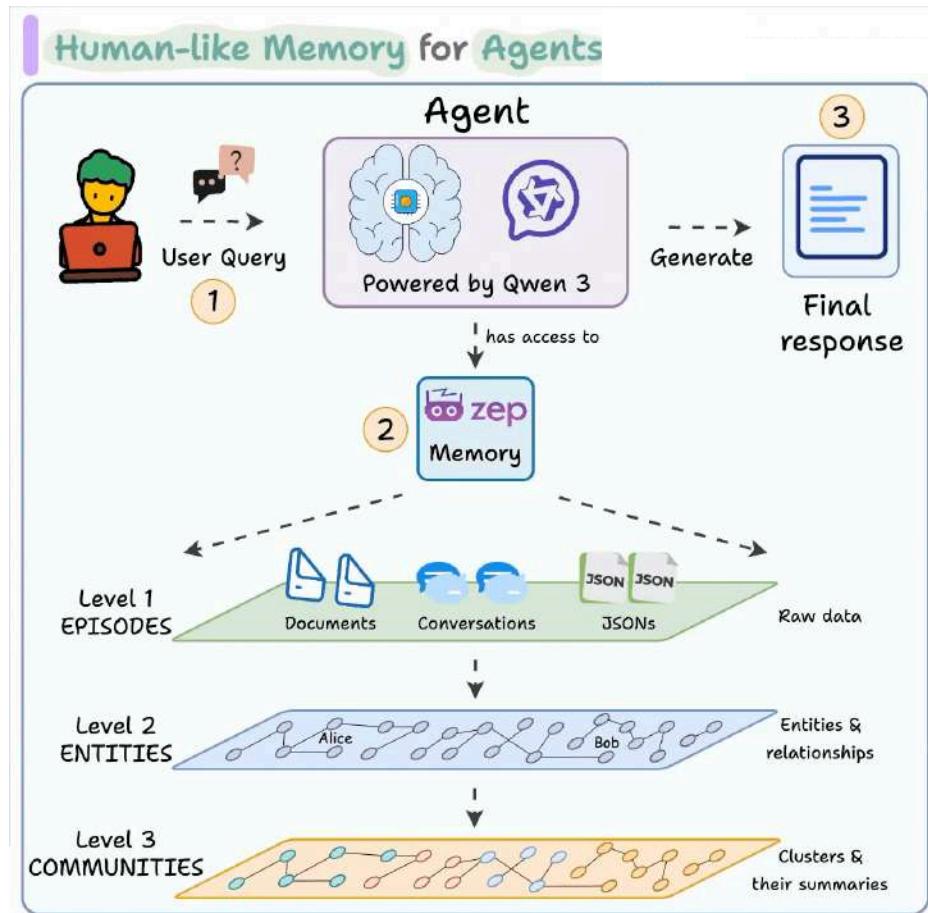


Done! Your deep research MCP server is live and connected to Cursor.



## #8) Human-like Memory for Agents

If a memory-less AI Agent is deployed in production, every interaction with the Agent will be a blank slate. Learn how to build an AI Agent with human-like memory to solve this.



Tech stack:

- Zep AI for the memory layer to AI agent
- Microsoft AutoGen for agent orchestration
- Ollama to locally serve Qwen3

Workflow:

- User submits a query

- Agent saves the conversation and extracts facts into memory
- Agent retrieves facts and summarizes
- Uses facts and history for informed responses

Let's implement this!

### #1) Setup LLM

We'll use a locally served Qwen 3 via Ollama. Check this out:

```
code2career
Set up LLM
ollama pull qwen3:4b
```

```
code2career
llm_config.py
Qwen3

# Ollama Model Configuration
config_list = [
    {
        "model": "qwen3:4b",
        "api_type": "ollama",
        "client_host": "http://127.0.0.1:11434", # Ollama host
    }
]
```

### #2) Initialise Zep Client

We're leveraging zep\_ai's Foundational Memory Layer to equip our Autogen agent with genuine task-completion capabilities.

```
code2career
Initialize Zep Client
zep
```

```
from zep_cloud.client import Zep

# Configure Zep
zep = Zep(api_key="YOUR_ZEP_API_KEY")
```

### #3) Create User Session

Create a Zep client session for the user, which the agent will use to manage memory. A user can have multiple sessions. Here's how it looks:

```

code2career
Create User Session
zep

import uuid
from zep_cloud.memory import FactRatingInstruction, FactRatingExamples

# User and session identifiers
user_name = "Cathy"
user_id = user_name + str(uuid.uuid4())[:4]
session_id = str(uuid.uuid4())

# Define fact rating instructions
fact_rating_instruction = """Rate facts by relevance and utility. Highly relevant facts directly impact the user's current needs or represent core preferences that affect multiple interactions. Low relevance facts are incidental details that rarely influence future conversations or decisions."""

fact_rating_examples = FactRatingExamples(
    high="The user is developing a Python application using the Streamlit framework.",
    medium="The user prefers dark mode interfaces when available.",
    low="The user mentioned it was raining yesterday."
)

# Add session for the user
zep.user.add(
    user_id=user_id,
    fact_rating_instruction=FactRatingInstruction(
        instruction=fact_rating_instruction,
        examples=fact_rating_examples,
    ),
)
zep.memory.add_session(user_id=user_id, session_id=session_id)
  
```

Zep Fact Rating Instructions

Zep Add Session

### #4) Define Zep Conversable Agent

Our Zep Memory Agent builds on Autogen's Conversable Agent, drawing live memory context from Zep Cloud with each user query.

It remains efficient by utilizing the session we just established.

Here's how it comes together:

```

from autogen import ConversableAgent, Agent
from zep_cloud.client import Zep
from zep_cloud import Message, Memory

class ZepConversableAgent(ConversableAgent): # Agent with Zep memory
    """A custom ConversableAgent that integrates with Zep for long-term memory."""

    def __init__(self,
                 name: str,
                 system_message: str,
                 llm_config: dict,
                 # ... other parameters ...
                 ):
        # Initialize the base ConversableAgent with standard parameters
        super().__init__(name=name, system_message=system_message, llm_config=llm_config, **kwargs)

        self.zep_session_id = zep_session_id
        self.zep_client = zep_client
        self.min_fact_rating = min_fact_rating
        self.original_system_message = system_message
        self.register_hook("process_message_before_send", self._zep_persist_assistant_messages)

    def _zep_persist_assistant_messages(self,
                                        message: Union[Dict, str],
                                        sender: Agent,
                                        recipient: Agent,
                                        silent: bool,
                                        ):
        """Agent sends a message to the user. Add the message to Zep."""
        if sender == self:
            content = message["content"] if isinstance(message, dict) else str(message)
            if content:
                self.zep_client.memory.add(
                    session_id=self.zep_session_id,
                    messages=[Message(role_type="assistant", role=self.name, content=content)]
                )
        return message

    def _zep_fetch_and_update_system_message(self):
        """Fetch facts and update system message."""
        memory = self.zep_client.memory.get(self.zep_session_id, min_rat)
        context = memory.context or "No specific facts recalled."
        self.update_system_message(
            f"{self.original_system_message}\n\nRelevant facts: {context}"
        )

    def _zep_persist_user_message(self, user_content: str, user_name: str = "User"):
        """User sends a message to the agent. Add the message to Zep."""
        if user_content:
            self.zep_client.memory.add(
                session_id=self.zep_session_id,
                messages=[Message(role_type="user", role=user_name, content=user_content)]
            )

```

**Inheriting Autogen Class**

**1. Persist Assistant Message**

**2. Update System Message**

**3. Persist User Message**

## #5) Setting up Agents

We initialize the Conversable Agent and a Stand-in Human Agent to manage chat interactions.

Here's the setup:

```

code2career
Setting up Agents
AG2

from zep_cloud.client import Zep
from autogen import ConversableAgent
from agent import ZepConversableAgent
from llm_config import config_list

# Create the autogen agent with Zep memory
agent = ZepConversableAgent(
    name="Zep Agent",
    system_message="""
    You are 'Zep Agent', a helpful and professional assistant. Support the user by remembering past interactions, offering accurate and clear answers, and maintaining a friendly, empathetic tone. Prioritize recent context, avoid repeating known info, and never reveal internal workings.
    """,
    llm_config={"config_list": config_list},
    zep_session_id=session_id,
    zep_client=zep,
    min_fact_ratings=0.7,
    function_map=None,
    human_input_mode="NEVER",
)

# Create Stand-in Human agent
user = ConversableAgent(
    name="Cathy",
    system_message="""
    You are a helpful mental health bot. You are seeking counsel from a mental health bot. Ask the bot about your previous conversation.
    """,
    llm_config={"config_list": config_list},
    human_input_mode="NEVER",
)

```

**1. Zep Conversable Agent**

**2. Stand-in Human Agent**

## #6) Handle Agentic Chat

The Zep Conversable Agent steps in to create a coherent, personalized response.

It seamlessly integrates memory and conversation. Here's how it works:

```

code2career
Handle Agentic Chat

result = agent.initiate_chat(
    user,
    message="Hi Cathy, nice to see you again. How are you doing today?",
    max_turns=2,
)

Cathy (to Cathy):
Hello, Cathie! I'm doing well, thank you for asking. I wanted to reflect on our previous conversation—do you remember what we discussed? It would be helpful to revisit that topic and

Cathy (to Cathie):
Of course, Cathie, we talked about your difficulty in processing your mother's passing and how you've been feeling down and unmotivated lately. It's completely natural to have these feelings.

Cathy (to Cathie):
Cathy: Thank you for continuing on, Cathie. Yes, I've been struggling with those feelings, and it's been tough to navigate. I'd like to explore some coping strategies to help me process

Cathy (to Cathie):
Absolutely, Cathie, here are some coping strategies to help you navigate your grief:
1. Journaling*: Writing your thoughts and feelings can be a great outlet and help you process your emotions.
2. Talk to someone**: Sharing your feelings with a trusted friend or therapist can provide support and understanding.
3. Mindfulness and meditation practices can help ground you and reduce stress and emotional tension.
4. Engage in a hobby: Doing something you enjoy, like reading, listening to music, or lighting a candle can foster connection and memory.
5. Physical activity*: Engaging in exercise can boost your mood and help alleviate stress.

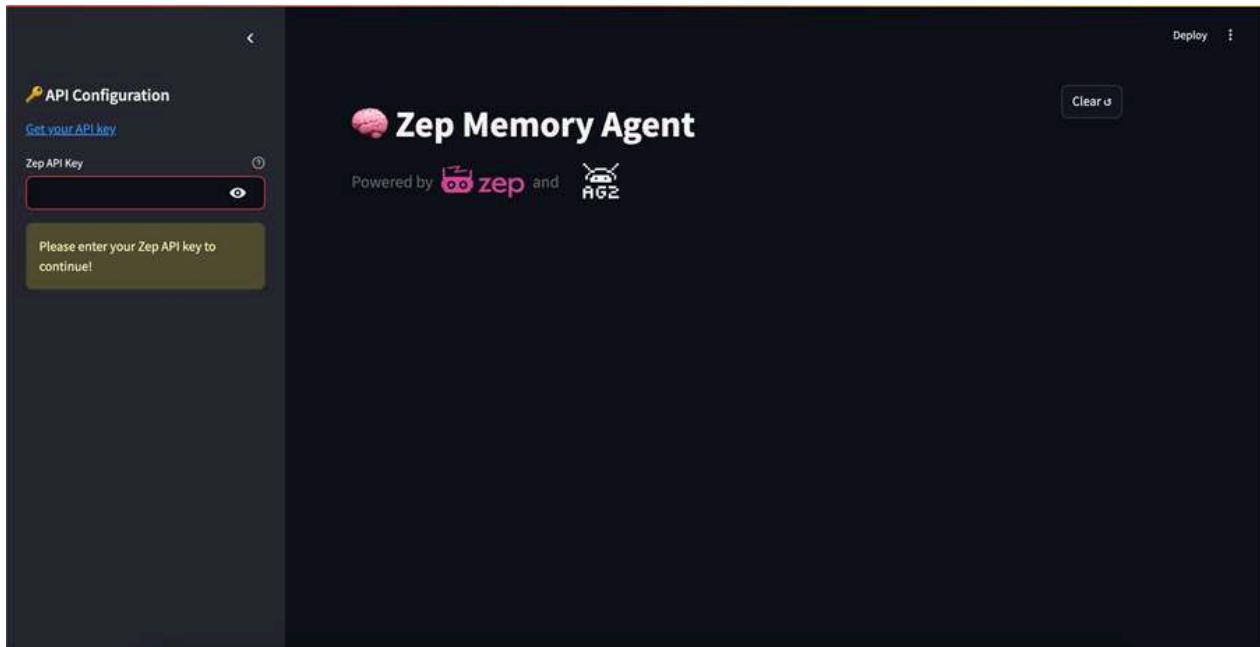
Remember, it's important to be gentle with yourself as you navigate this process; what resonates with you

```

## #7) Streamlit UI

We created a streamlined Streamlit UI to ensure smooth and simple interactions with the Agent.

Here's what it looks like:

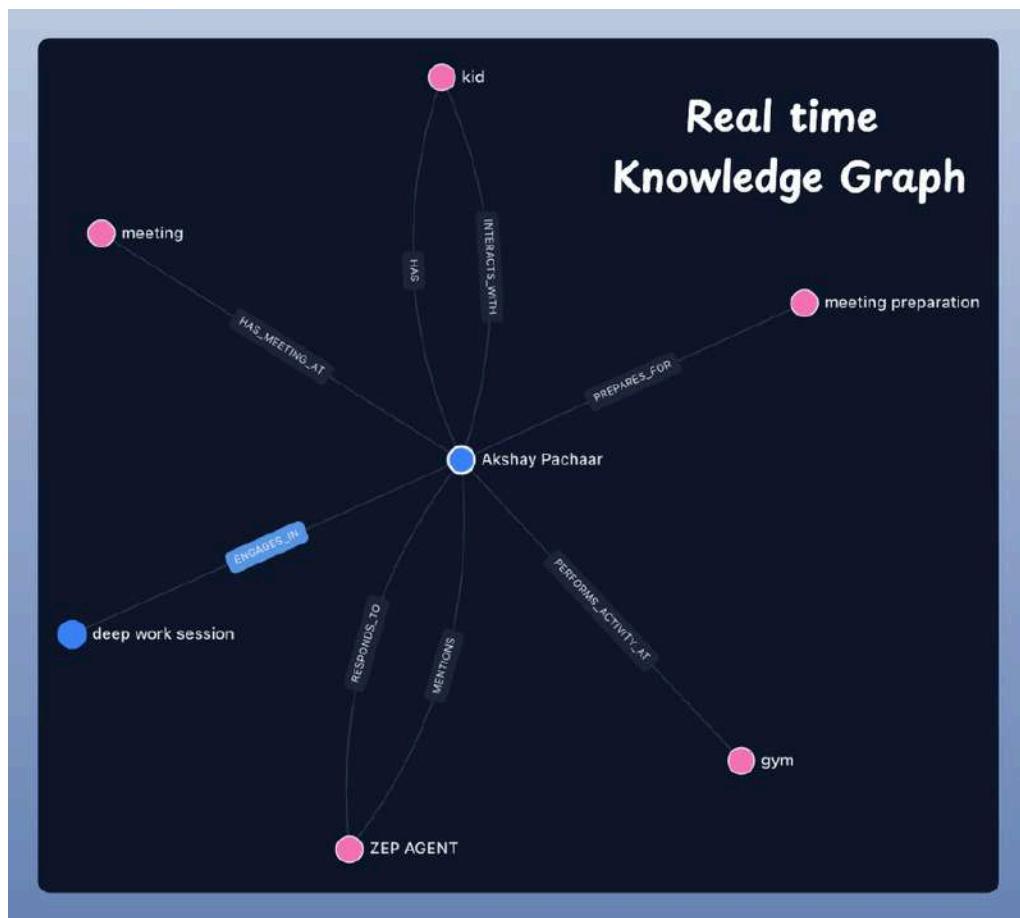


## #8) Visualize Knowledge Graph

We can interactively map users' conversations across multiple sessions with Zep Cloud's UI.

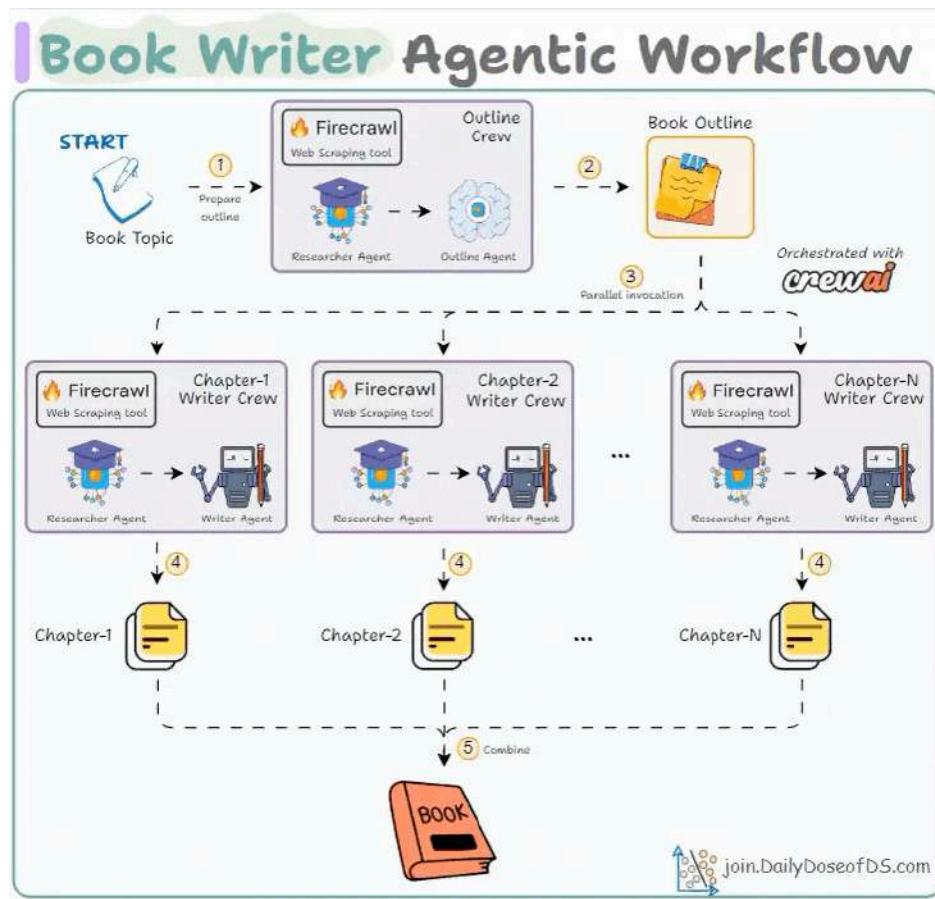
This powerful tool allows us to visualize how knowledge evolves through a graph.

Take a look:



## #9) Multi-agent Book Writer

Build an Agentic workflow that writes a 20k-word book from a 3-5 word book title.



Tech stack:

- Firecrawl for web scraping.
- CrewAI for orchestration.
- Ollama to serve Qwen 3 locally.
- LightningAI for development and hosting

Workflow:

- Using Firecrawl, Outline Crew scrapes data related to the book title and decides the chapter count and titles.
- Multiple writer Crews work in parallel to write one chapter each.
- Combine all chapters to get the book.

Let's implement this!

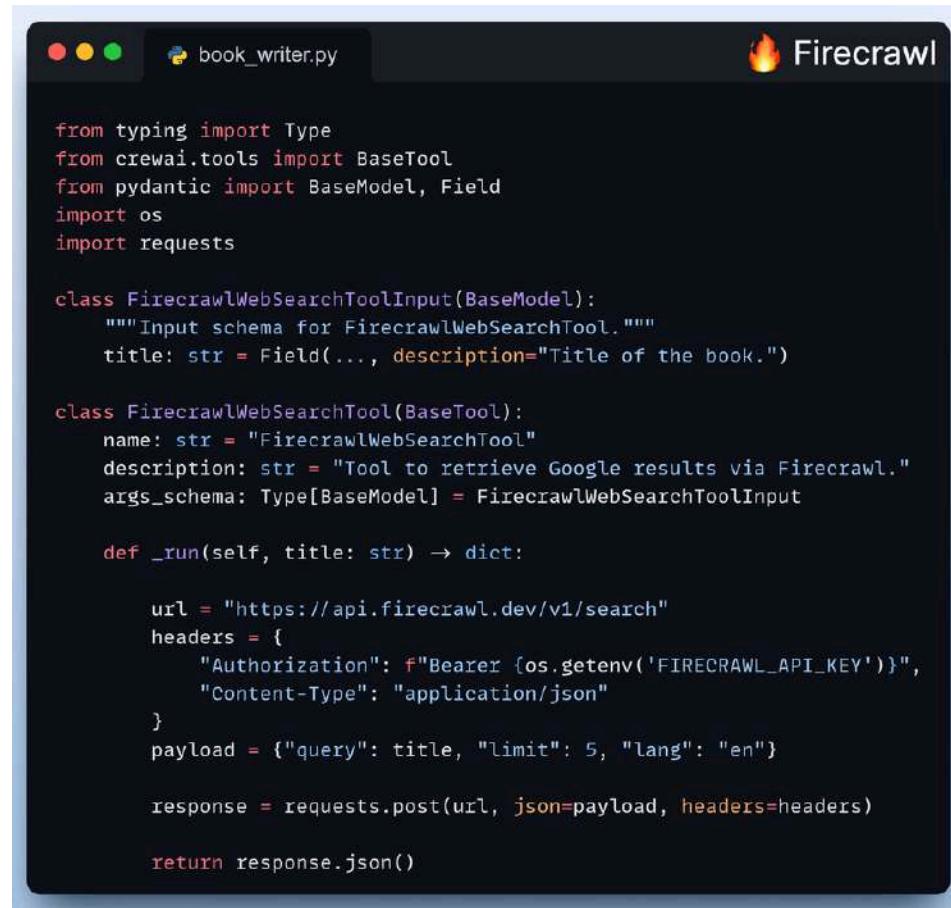
#### #1) Scraping tool - SERP API

Books demand research. Thus, we'll use Firecrawl's SERP API to scrape data.

Tool usage:

- Outline Crew → to research the book title and prepare an outline.
- Writer Crew → to research the chapter title and write it.

See this code:



```
from typing import Type
from crewai.tools import BaseTool
from pydantic import BaseModel, Field
import os
import requests

class FirecrawlWebSearchToolInput(BaseModel):
    """Input schema for FirecrawlWebSearchTool."""
    title: str = Field(..., description="Title of the book.")

class FirecrawlWebSearchTool(BaseTool):
    name: str = "FirecrawlWebSearchTool"
    description: str = "Tool to retrieve Google results via Firecrawl."
    args_schema: Type[BaseModel] = FirecrawlWebSearchToolInput

    def _run(self, title: str) -> dict:
        url = "https://api.firecrawl.dev/v1/search"
        headers = {
            "Authorization": f"Bearer {os.getenv('FIRECRAWL_API_KEY')}",
            "Content-Type": "application/json"
        }
        payload = {"query": title, "limit": 5, "lang": "en"}

        response = requests.post(url, json=payload, headers=headers)

        return response.json()
```

## #2) Setup Qwen 3 locally

We'll serve Qwen 3 locally through Ollama. To do this:

- First, we download it locally.
- Next, we define it with the CrewAI's LLM class.



### #3) Outline Crew

This Crew has two Agents:

- Research Agent → Uses the Firecrawl scraping tool to scrape data related to the book's title and prepare insights.
- Outline Agent → Uses the insights to output total chapters and titles as Pydantic output.



```

from crewai import Agent, Crew, Process, Task

class Outline(BaseModel):
    total_chapters: int
    titles: list[str]

    Pydantic output
    for outline

research_agent = Agent(role="Book Research Agent",
                       goal="""Research the topic {topic} and
                               collect information about it.""",
                       backstory="""You are an {topic} expert and have
                               a deep understanding of it.""",
                       tools=[FirecrawlWebSearchTool()])

research_task = Task(description="""Prepare insights and key points that
                                      to create an outline for a book""",
                     expected_output="Insights about {topic}.",
                     agent=research_agent)

outline_agent = Agent(role="Book Outline Writer",
                      goal="""Generate outline of book about {topic}""",
                      backstory="""You are an {topic} expert and
                               have a deep understanding of it""")

outline_task = Task(description="Write outline for a book about {topic}",
                     expected_output="Total chapter and titles",
                     agent=outline_agent,
                     output_pydantic=Outline)

OutlineCrew = Crew(agents=[research_agent, outline_agent],
                  tasks=[research_task, outline_task],
                  process=ProcessSEQUENTIAL)

```

#### #4) Writer Crew

This Crew has two Agents:

- Research Agent → Uses the Firecrawl Scraping tool to scrape data related to a chapter's title and prepare insights.
- Write Agent → Uses the insights to write a chapter.

Check this code:



```

from crewai import Agent, Crew, Process, Task

class Chapter(BaseModel):
    title: str
    content: str

    Pydantic output
    for chapter

research_agent = Agent(role="Topic Researcher"
                       goal="""Research the topic {title} and
                               collect information about it.""",
                       backstory="""You are an {title} expert and have
                               a deep understanding of it""",
                       tools=[FirecrawlWebSearchTool()])

research_task = Task(description="""Prepare insights about {title}
                                    for the book {topic}""",
                     expected_output="Insights about {title}.",
                     agent=research_agent)

writer_agent = Agent(role="Senior Writer"
                     goal="""Write a chapter about {title}""",
                     backstory="""You are an {topic} expert and
                     have a deep understanding of it""")

writer_task = Task(description="Write a chapter about {title}",
                    expected_output="A chapter about {title}",
                    agent=writer_agent,
                    output_pydantic=Chapter)

OutlineCrew = Crew(agents=[research_agent, writer_agent],
                  tasks=[research_task, writer_task],
                  process=ProcessSEQUENTIAL)

```

## #5) Create a Flow

We use CrewAI Flows to orchestrate the workflow.

First, the outline method invokes the Outline Crew, which:

- research the topic using the scraping tool.
- returns the total number of chapters and the corresponding titles.

This is implemented below:

```
from crewai.flow import Flow, listen, start

class Chapter(BaseModel):
    title: str = ""
    content: str = ""

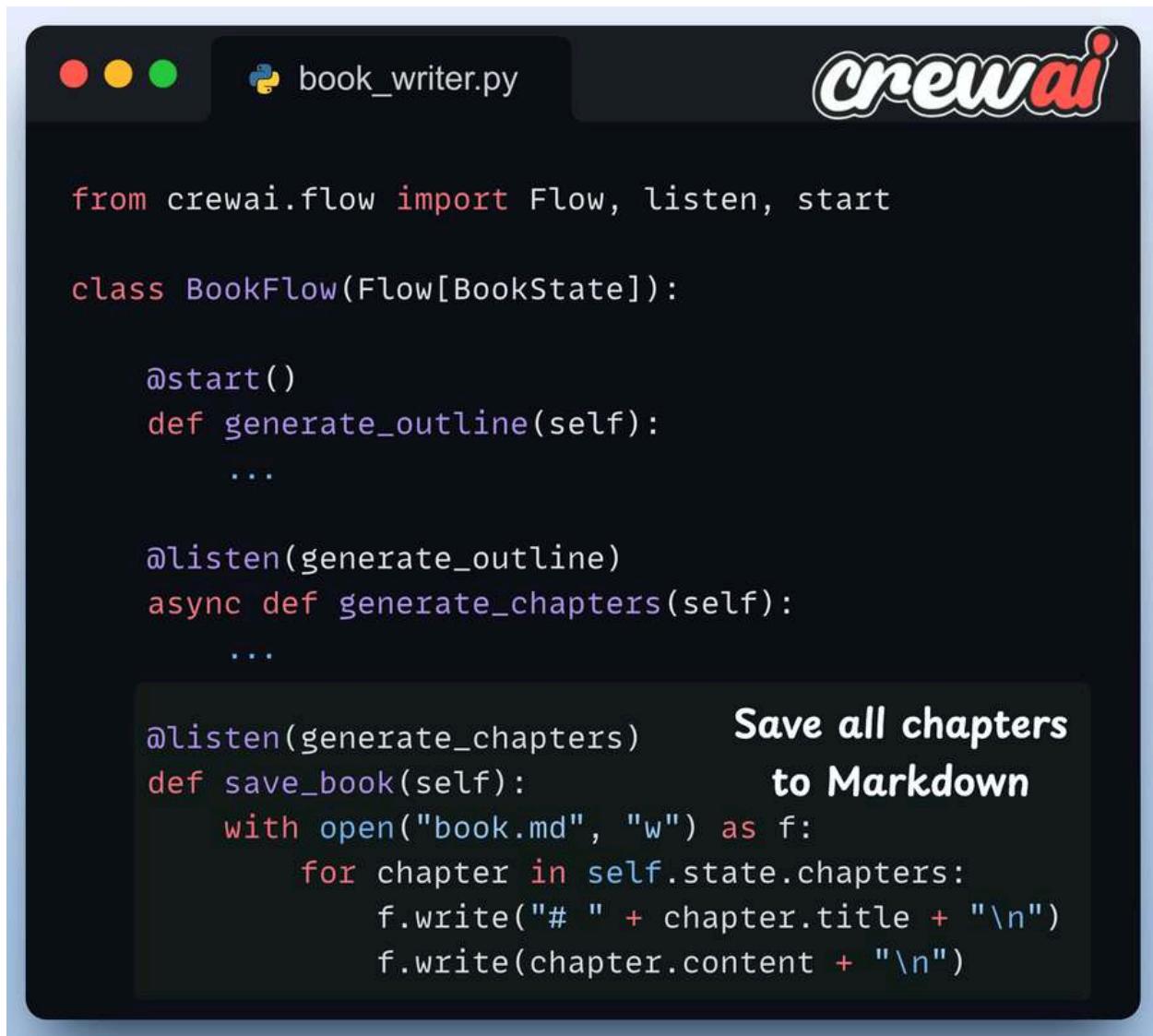
class BookState(BaseModel):
    topic: str = "Astronomy in 2025"
    total_chapters: int = 0
    titles: list[str] = []
    chapters: list[Chapter] = []

class BookFlow(Flow[BookState]):  
    @start()  
    def generate_outline(self):  
        outline = (OutlineCrew()  
            .crew()  
            .kickoff(inputs={"topic": self.state.topic})  
        )  
        self.state.total_chapters = outline.pydantic.total_chapters  
        self.state.titles = outline.pydantic.titles
```

## #6) Save the book

Once all Writer Crews have finished execution, we save the book as a Markdown file.

Check this code:



```
from crewai.flow import Flow, listen, start

class BookFlow(Flow[BookState]):

    @start()
    def generate_outline(self):
        ...

    @listen(generate_outline)
    async def generate_chapters(self):
        ...

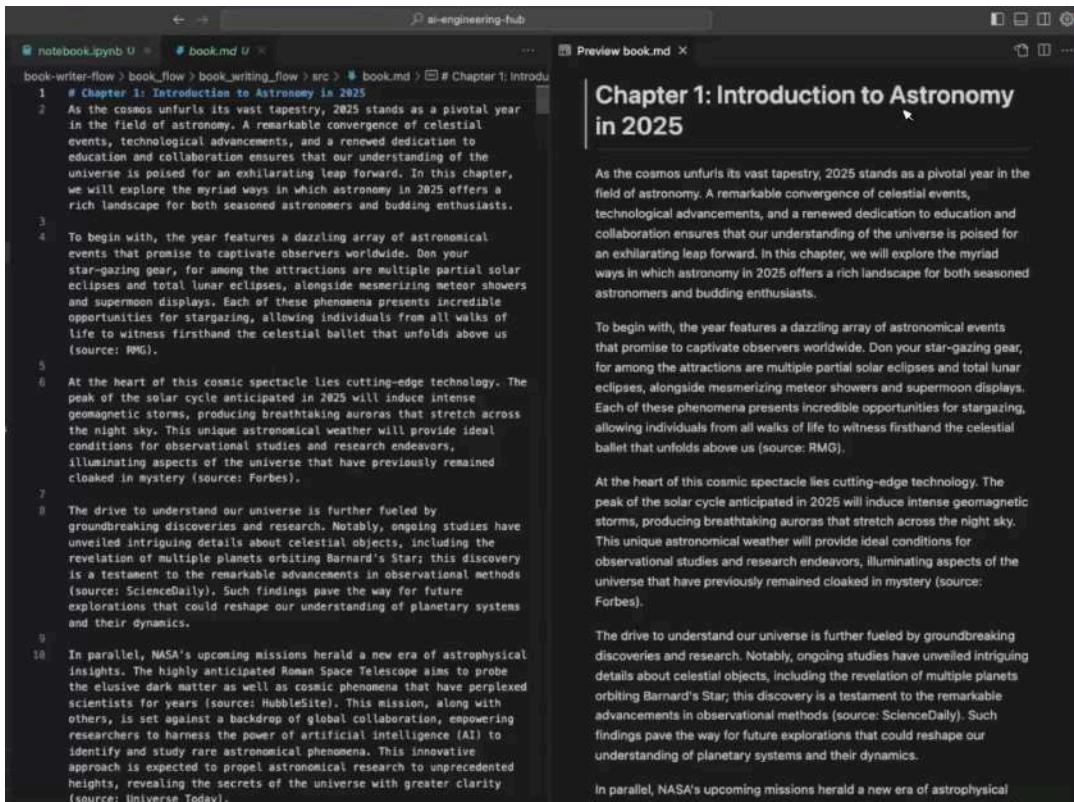
    @listen(generate_chapters)
    def save_book(self):
        with open("book.md", "w") as f:
            for chapter in self.state.chapters:
                f.write("# " + chapter.title + "\n")
                f.write(chapter.content + "\n")
```

## #7) Kickoff the Flow

Finally, we run the Flow.

- First, the Outline Crew is invoked. It utilizes the Firecrawl scraping tool to prepare the outline.
- Next, many Writer Crews are invoked in parallel to write one chapter each.

This workflow runs for ~2 minutes, and we get a neatly written book about the specified topic—"Astronomy in 2025." Here's the book our Agentic workflow wrote:



The screenshot shows two windows side-by-side. On the left is a Jupyter Notebook cell titled 'book\_writer-flow > book\_flow > book\_writing\_flow > src > book.md' containing the source code for the book. On the right is a preview window titled 'Preview book.md' showing the rendered content of the book.

```

1 # Chapter 1: Introduction to Astronomy in 2025
2 As the cosmos unfurls its vast tapestry, 2025 stands as a pivotal year
   in the field of astronomy. A remarkable convergence of celestial
   events, technological advancements, and a renewed dedication to
   education and collaboration ensures that our understanding of the
   universe is poised for an exhilarating leap forward. In this chapter,
   we will explore the myriad ways in which astronomy in 2025 offers a
   rich landscape for both seasoned astronomers and budding enthusiasts.
3
4 To begin with, the year features a dazzling array of astronomical
   events that promise to captivate observers worldwide. Don your
   star-gazing gear, for among the attractions are multiple partial solar
   eclipses and total lunar eclipses, alongside mesmerizing meteor showers
   and supermoon displays. Each of these phenomena presents incredible
   opportunities for stargazing, allowing individuals from all walks of
   life to witness firsthand the celestial ballet that unfolds above us
   (source: RMG).
5
6 At the heart of this cosmic spectacle lies cutting-edge technology. The
   peak of the solar cycle anticipated in 2025 will induce intense
   geomagnetic storms, producing breathtaking auroras that stretch across
   the night sky. This unique astronomical weather will provide ideal
   conditions for observational studies and research endeavors,
   illuminating aspects of the universe that have previously remained
   cloaked in mystery (source: Forbes).
7
8 The drive to understand our universe is further fueled by
   groundbreaking discoveries and research. Notably, ongoing studies have
   unveiled intriguing details about celestial objects, including the
   revelation of multiple planets orbiting Barnard's Star; this discovery
   is a testament to the remarkable advancements in observational methods
   (source: ScienceDaily). Such findings pave the way for future
   explorations that could reshape our understanding of planetary systems
   and their dynamics.
9
10 In parallel, NASA's upcoming missions herald a new era of astrophysical
    insights. The highly anticipated Roman Space Telescope aims to probe
    the elusive dark matter as well as cosmic phenomena that have perplexed
    scientists for years (source: HubbleSite). This mission, along with
    others, is set against a backdrop of global collaboration, empowering
    researchers to harness the power of artificial intelligence (AI) to
    identify and study rare astronomical phenomena. This innovative
    approach is expected to propel astronomical research to unprecedented
    heights, revealing the secrets of the universe with greater clarity
    (source: Universe Today).

```

**Chapter 1: Introduction to Astronomy in 2025**

As the cosmos unfurls its vast tapestry, 2025 stands as a pivotal year in the field of astronomy. A remarkable convergence of celestial events, technological advancements, and a renewed dedication to education and collaboration ensures that our understanding of the universe is poised for an exhilarating leap forward. In this chapter, we will explore the myriad ways in which astronomy in 2025 offers a rich landscape for both seasoned astronomers and budding enthusiasts.

To begin with, the year features a dazzling array of astronomical events that promise to captivate observers worldwide. Don your star-gazing gear, for among the attractions are multiple partial solar eclipses and total lunar eclipses, alongside mesmerizing meteor showers and supermoon displays. Each of these phenomena presents incredible opportunities for stargazing, allowing individuals from all walks of life to witness firsthand the celestial ballet that unfolds above us (source: RMG).

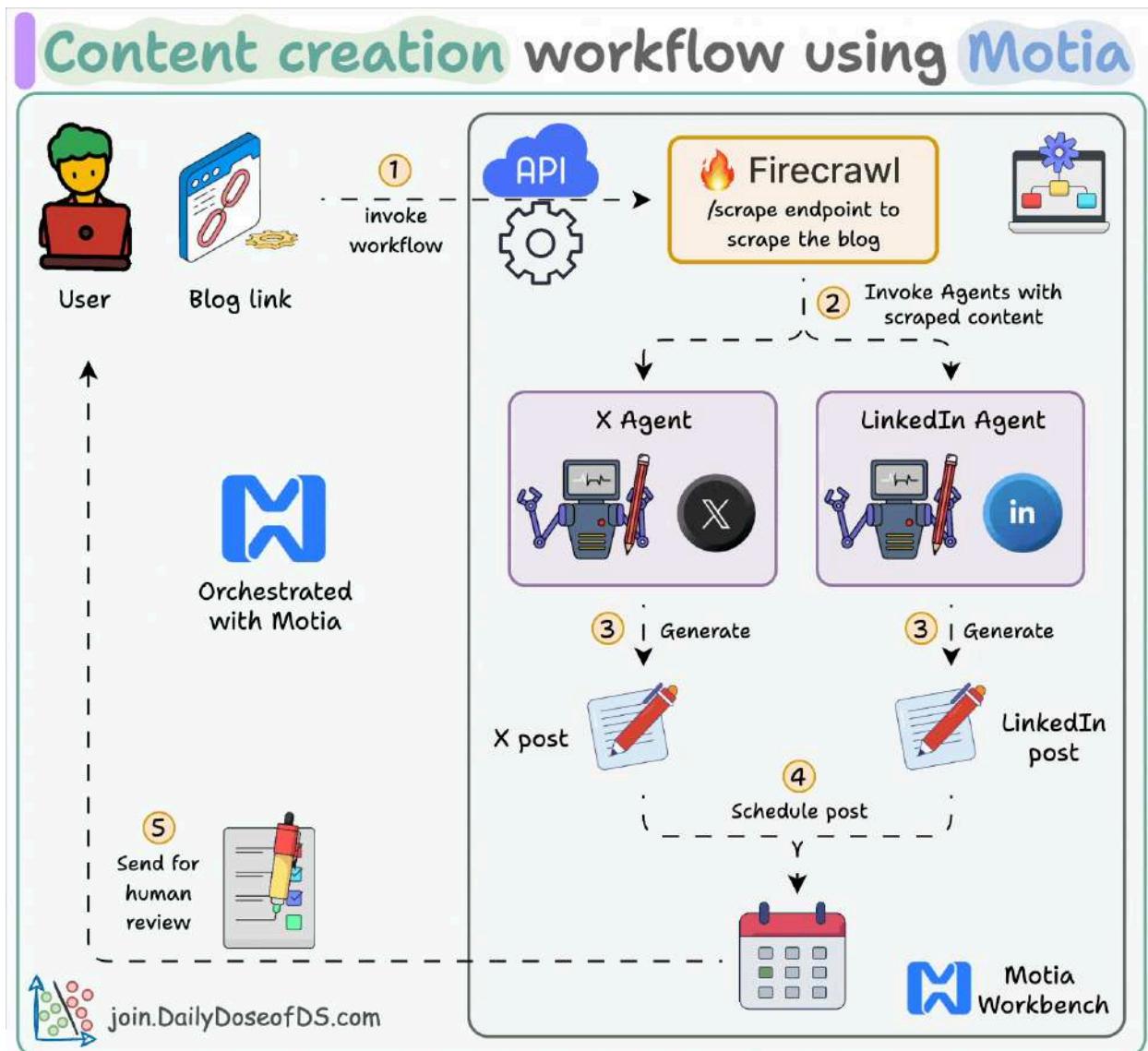
At the heart of this cosmic spectacle lies cutting-edge technology. The peak of the solar cycle anticipated in 2025 will induce intense geomagnetic storms, producing breathtaking auroras that stretch across the night sky. This unique astronomical weather will provide ideal conditions for observational studies and research endeavors, illuminating aspects of the universe that have previously remained cloaked in mystery (source: Forbes).

The drive to understand our universe is further fueled by groundbreaking discoveries and research. Notably, ongoing studies have unveiled intriguing details about celestial objects, including the revelation of multiple planets orbiting Barnard's Star; this discovery is a testament to the remarkable advancements in observational methods (source: ScienceDaily). Such findings pave the way for future explorations that could reshape our understanding of planetary systems and their dynamics.

In parallel, NASA's upcoming missions herald a new era of astrophysical

## #10) Multi-agent Content Creation System

Build an Agentic workflow that turns any URL into social media posts and auto-schedules them via Typefully.



Tech stack:

- Motia as the unified backend framework
- Firecrawl to scrape web content
- Ollama to locally serve Deepseek-R1 LLM

Workflow:

- User submits URL to scrape
- Firecrawl scrapes content and converts it to markdown
- Twitter and LinkedIn agents run in parallel to generate content
- Generated content gets scheduled via Typefully

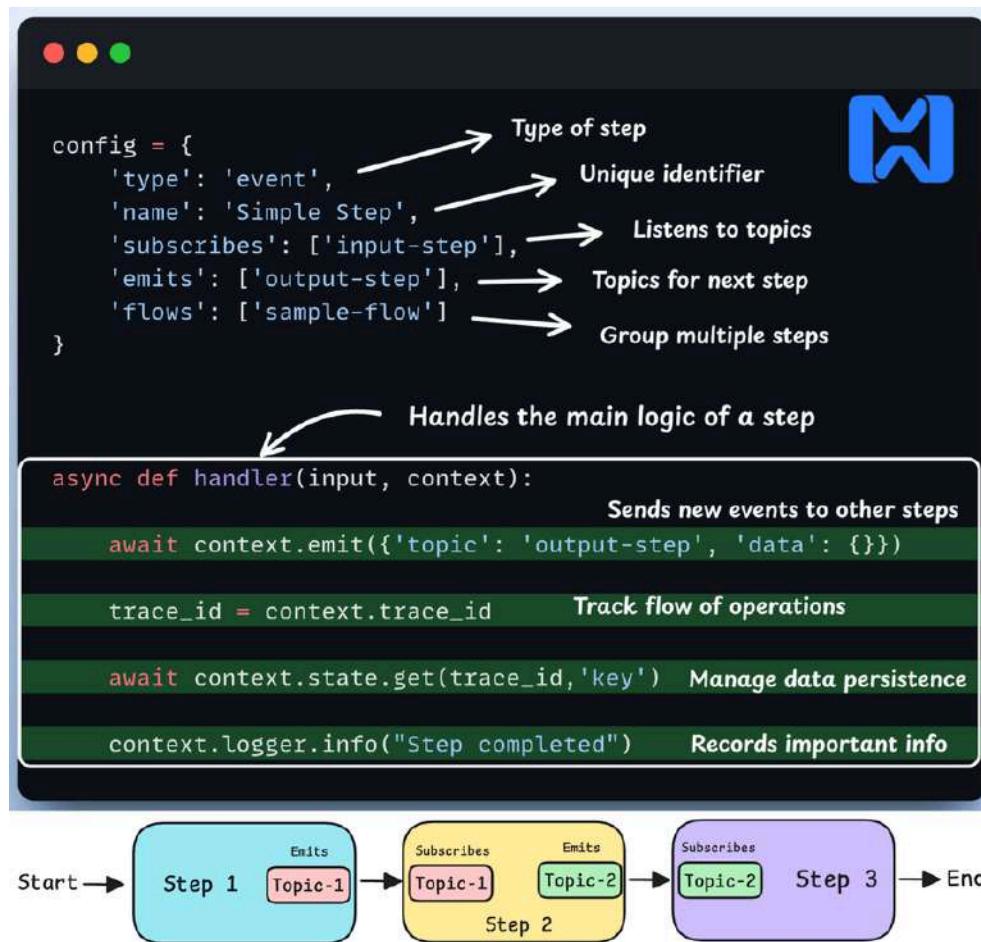
Let's implement this:

Steps are the fundamental building blocks of Motia.

They consist of two main components:

- The Config object: It instructs Motia on how to interact with a step.
- The handler function: It defines the main logic of a step.

Check this out:



With that understanding in mind, let's start building our content creation workflow.

### #1) Entry point (API)

We start our content generation workflow by defining an API step that takes in a URL from the user via a POST request.

Check this out:



```
config = {
    'type': 'api',
    'name': 'ContentGenerationAPI',
    'description': 'Triggers content generation from URL',
    'path': '/generate-content',
    'method': 'POST',
    'emits': ['scrape-article'],
    'flows': ['content-generation']
}

async def handler(req, context):
    await context.emit({
        'topic': 'scrape-article',
        'data': {
            'requestId': context.traceId,
            'url': req['body']['url']
        }
    })

    return {
        'status': 200,
        'body': {
            'message': 'Content generation started',
            'requestId': context.traceId,
            'url': req['body']['url'],
            'status': 'processing'
        }
    }
```

Workflow initiated successfully

## #2) Web scraping

This step scrapes the article content using Firecrawl and emits the next step in the workflow.

Steps can be connected together in a sequence, where the output of one step becomes the input for another.

Check this out:



```

from firecrawl import FirecrawlApp

config = {
    'type': 'event',
    'name': 'ScrapeArticle',
    'description': 'Scrapes content using Firecrawl',
    'subscribes': ['scrape-article'],
    'emits': ['generate-content'],
    'input': ScrapeInput,
    'flows': ['content-generation']
}

async def handler(input, context):
    context.logger.info(f"Scraping article: {input['url']}")

    app = FirecrawlApp(api_key=FIRECRAWL_API_KEY)
    scrapeResult = await app.scrape_url(input['url'])

    await context.emit({
        'topic': 'generate-content',           Sent as input
        'data': {                            to the next step
            'requestId': input['requestId'],
            'url': str(input['url']),
            'title': scrapeResult.metadata.get('title'),
            'content': scrapeResult.markdown,
        }
    })

```

### #3) Content generation

The scraped content gets fed to the X and LinkedIn agents that run in parallel and generate curated posts.

We define all our prompting and AI logic in the handler that runs automatically when a step is triggered.

Check this out:

```

import asyncio
from ollama import AsyncClient

config = {
    'type': 'event',
    'name': 'GenerateContent',
    'description': 'Generates social media content',
    'subscribes': ['generate-content'],
    'emits': ['schedule-content'],
    'input': GenerateInput,
    'flows': ['content-generation']
}

async def handler(input, context):
    twitter_resp, linkedin_resp = await asyncio.gather(
        AsyncClient().chat(model="deepseek-r1",
                           messages=[{
                               'role': 'user',
                               'content': twitter_prompt
                           }]),
        AsyncClient().chat(model="deepseek-r1",
                           messages=[{
                               'role': 'user',
                               'content': linkedin_prompt
                           }])
    )

    await context.emit({
        'topic': 'schedule-content',
        'data': { 'content': {twitter_content, linkedin_content} }
    })

```

**Both agents invoked  
in parallel**

X Agent

LinkedIn Agent

#### #4) Scheduling

After the content is generated we draft it in Typefully where we can easily review our social media posts.

Motia also allows us to mix and match different languages within the same workflow providing great flexibility.

Check this typescript code:



```
import { EventConfig, Handlers } from 'motia'
import axios from 'axios'

export const config: EventConfig = {
  type: 'event',
  name: 'Schedule',
  description: 'Schedules social media posts using Typefully',
  subscribes: ['schedule-content'],
  emits: [],
  flows: ['content-generation']
}

export const handler: Handlers['Schedule'] = async (input, {logger}) => {
  const typefullyHeaders = {
    'X-API-KEY': `Bearer ${TYPEFULLY_API_KEY}`,
    'Content-Type': 'application/json'
  }

  await axios.post('https://api.typefully.com/v1/drafts/', {
    content: twitterTweets.join('\n\n\n\n'),
    schedule_date: twitterScheduleTime
  }, { headers: typefullyHeaders })

  await axios.post('https://api.typefully.com/v1/drafts/', {
    content: input.content.linkedin.post,
    schedule_date: linkedinScheduleTime,
  }, { headers: typefullyHeaders })

  logger.info(`Content scheduling completed successfully!\n`)
}
```

After defining our steps, we install required dependencies using `npm install` and run the Motia workbench using `npm run dev` commands.

Check this out:

The screenshot shows a Mac OS X terminal window with the title bar 'command line'. Inside, two commands are shown with arrows pointing to their descriptions:

- \$ npm install → **Install dependencies**
- \$ npx motia dev → **Run development server**

A curved arrow points from the 'Run development server' description down to the terminal log area. The log output is as follows:

```
> dev
> . python_modules/bin/activate && motia dev

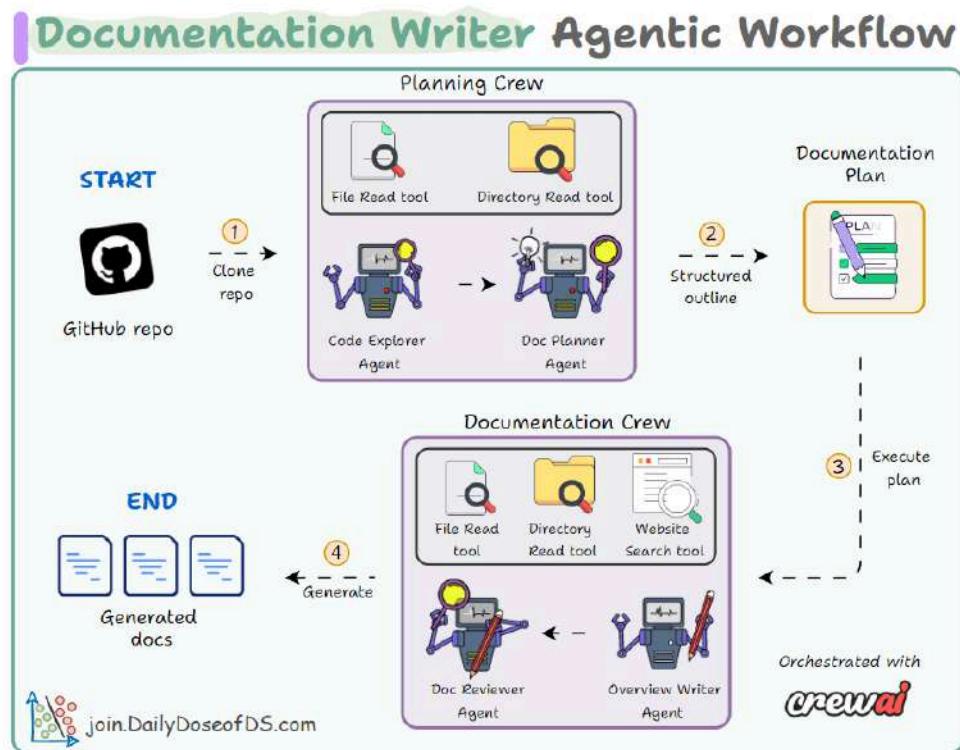
Activating Python environment...
→ [CREATED] Flow content-generation created
→ [CREATED] Step (Event) steps/scrape.step.py created
→ [CREATED] Step (Event) steps/schedule.step.ts created
→ [CREATED] step (Event) steps/generate.step.py created
→ [CREATED] Step (API) steps/api.step.py created
⚡ Server ready and listening on port 3000
🔗 Open http://localhost:3000/ to open workbench ✨
```

Motia workbench provides an interactive UI to help build, monitor and debug our flows.

With one-click you can also deploy it to the cloud!

## #11) Documentation Writer Flow

Build an Agentic workflow that generates full project documentation from just a GitHub repo URL.



Tech stack:

- CrewAI for multi-agent orchestration
- Ollama to locally serve DeepSeek-R1 LLM

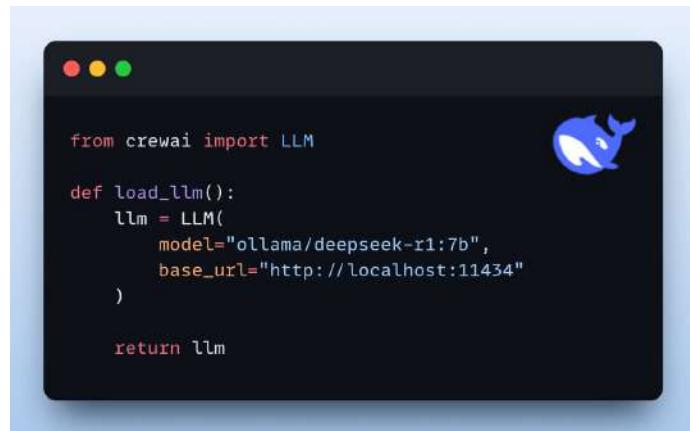
Workflow:

- User specifies GitHub repo
- Planning crew creates documentation plan
- Documentation crew writes documentation according to plan
- Generated docs get saved to local directory

Let's implement this!

## #1) Setup LLM

We will use Deepseek-R1 as the LLM, served locally using Ollama.



```
from crewai import LLM

def load_llm():
    llm = LLM(
        model="ollama/deepseek-r1:7b",
        base_url="http://localhost:11434"
    )

    return llm
```

## #2) Define Pydantic schema

We define the following pydantic schemas for robust structured outputs.

This ensures data validation and integrity before generating the documentation files. Check this code:



```
from pathlib import Path
from pydantic import BaseModel

# Define data structures for doc planning output
class DocItem(BaseModel):
    """Represents a documentation item"""
    title: str
    description: str
    prerequisites: str
    examples: list[str]
    goal: str

class DocPlan(BaseModel):
    """Documentation plan outline"""
    overview: str
    docs: list[DocItem]

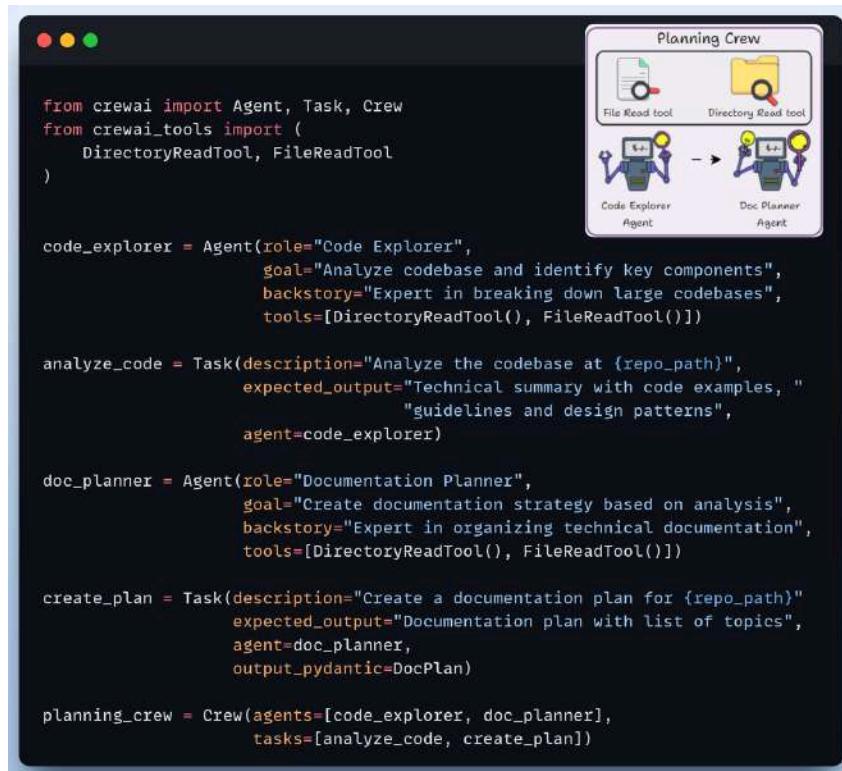
class DocumentationState(BaseModel):
    """State for the documentation flow"""
    project_url: str = ""
    repo_path: Path = "workdir/"
    docs: list[str] = []
```

## #3) Planning Crew

This crew oversees strategy for the outline via:

- Code explorer agent -> Analyzes codebase for key components, patterns and relationships.
- Doc planner agent -> Creates outline based on codebase analysis as pydantic output.

Check this out:



```

from crewai import Agent, Task, Crew
from crewai_tools import (
    DirectoryReadTool, FileReadTool
)

code_explorer = Agent(role="Code Explorer",
                      goal="Analyze codebase and identify key components",
                      backstory="Expert in breaking down large codebases",
                      tools=[DirectoryReadTool(), FileReadTool()])

analyze_code = Task(description="Analyze the codebase at {repo_path}",
                     expected_output="Technical summary with code examples, "
                     "guidelines and design patterns",
                     agent=code_explorer)

doc_planner = Agent(role="Documentation Planner",
                     goal="Create documentation strategy based on analysis",
                     backstory="Expert in organizing technical documentation",
                     tools=[DirectoryReadTool(), FileReadTool()])

create_plan = Task(description="Create a documentation plan for {repo_path}"
                    expected_output="Documentation plan with list of topics",
                    agent=doc_planner,
                    output_pydantic=DocPlan)

planning_crew = Crew(agents=[code_explorer, doc_planner],
                      tasks=[analyze_code, create_plan])

```

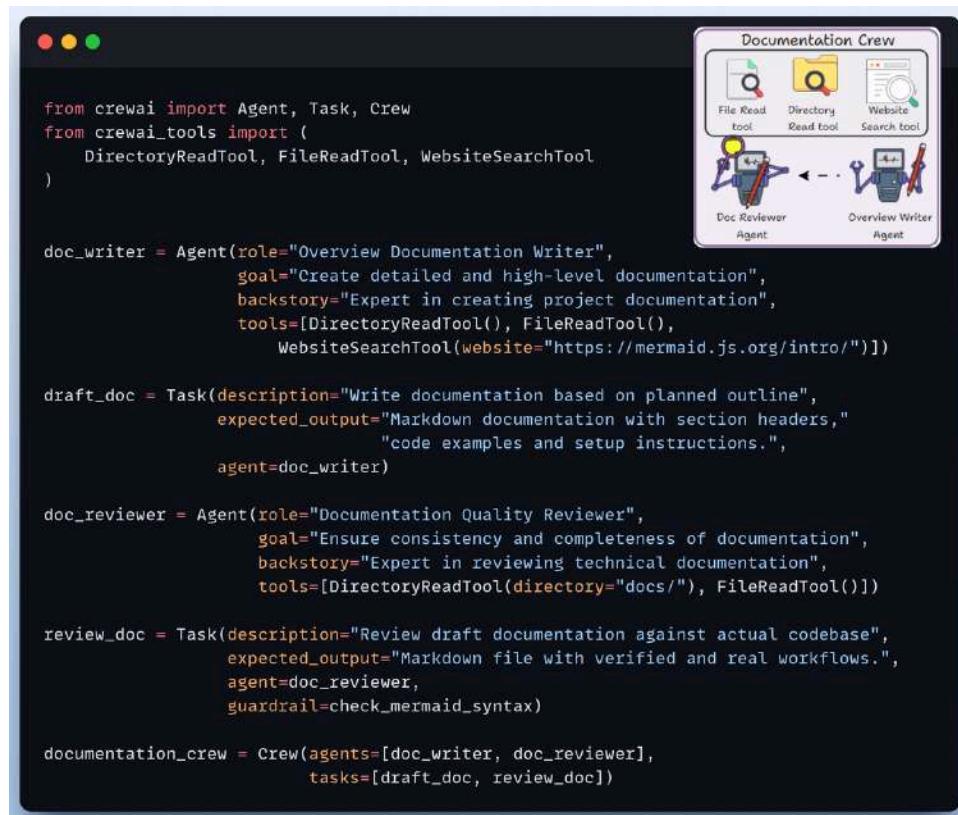
## #4) Documentation Crew

This crew writes and reviews the documentation via:

- Doc writer agent -> Generates a high-level draft based on the planned outline.

- Doc reviewer agent -> Reviews the draft for consistency, accuracy and completeness.

Check this out:



```

from crewai import Agent, Task, Crew
from crewai_tools import (
    DirectoryReadTool, FileReadTool, WebsiteSearchTool
)

doc_writer = Agent(role="Overview Documentation Writer",
                    goal="Create detailed and high-level documentation",
                    backstory="Expert in creating project documentation",
                    tools=[DirectoryReadTool(), FileReadTool(),
                           WebsiteSearchTool(website="https://mermaid.js.org/intro/")])

draft_doc = Task(description="Write documentation based on planned outline",
                  expected_output="Markdown documentation with section headers,",
                  "code examples and setup instructions.",
                  agent=doc_writer)

doc_reviewer = Agent(role="Documentation Quality Reviewer",
                      goal="Ensure consistency and completeness of documentation",
                      backstory="Expert in reviewing technical documentation",
                      tools=[DirectoryReadTool(directory="docs/"), FileReadTool()])

review_doc = Task(description="Review draft documentation against actual codebase",
                  expected_output="Markdown file with verified and real workflows.",
                  agent=doc_reviewer,
                  guardrail=check_mermaid_syntax)

documentation_crew = Crew(agents=[doc_writer, doc_reviewer],
                          tasks=[draft_doc, review_doc])

```

## #5) Create Documentation Flow

After setting up our crews, we create the main workflow that:

- Clones the GitHub repo
- Plans and saves the outline
- Generates documentation based on the outline
- Saves final docs to local directory

Check this code:

```

import subprocess
from crewai.flow.flow import Flow, listen, start

class CreateDocumentationFlow(Flow[DocumentationState]):
    @start()
    def clone_repo(self):
        """Clone the GitHub repository"""
        subprocess.run(["git", "clone", self.state.project_url, self.state.repo_path])
        return self.state

    @listen(clone_repo)
    def plan_docs(self):
        """Creates documentation outline"""
        result = planning_crew.kickoff(inputs={'repo_path': self.state.repo_path})
        return result

    @listen(plan_docs)
    def save_plan(self, plan):
        """Saves the planned outline for next crew"""
        with open("docs/plan.json", "w") as f:
            f.write(plan.raw)

    @listen(plan_docs)
    def create_docs(self, plan):
        """Generate and save final docs to local directory"""
        for doc in plan.pydantic.docs:
            result = documentation_crew.kickoff(inputs={...})

            with open(f"docs/{title}", "w") as f:
                f.write(result.raw)
        return self.state.docs

```

## #6) Kickoff the flow

Finally, when we have everything ready, we kick off our documentation flow with the GitHub repo URL.

Check this out:

The screenshot shows a terminal window with the following content:

```
repo_url = "https://github.com/username/repo"

flow = CreateDocumentationFlow()

flow.kickoff(inputs={"project_url": repo_url})
```

A callout box highlights the line `flow.kickoff(inputs={"project_url": repo_url})`. A curved arrow points from this line down to the terminal output below, which shows the flow execution logs.

Flow Execution

```
Starting Flow Execution
Name: CreateDocumentationFlow
ID: 0f54b988-af69-4dfb-9433-158f3feeb42f

Flow: CreateDocumentationFlow
ID: 0f54b988-af69-4dfb-9433-158f3feeb42f
└ Starting Flow...
  └ running: clone_repo

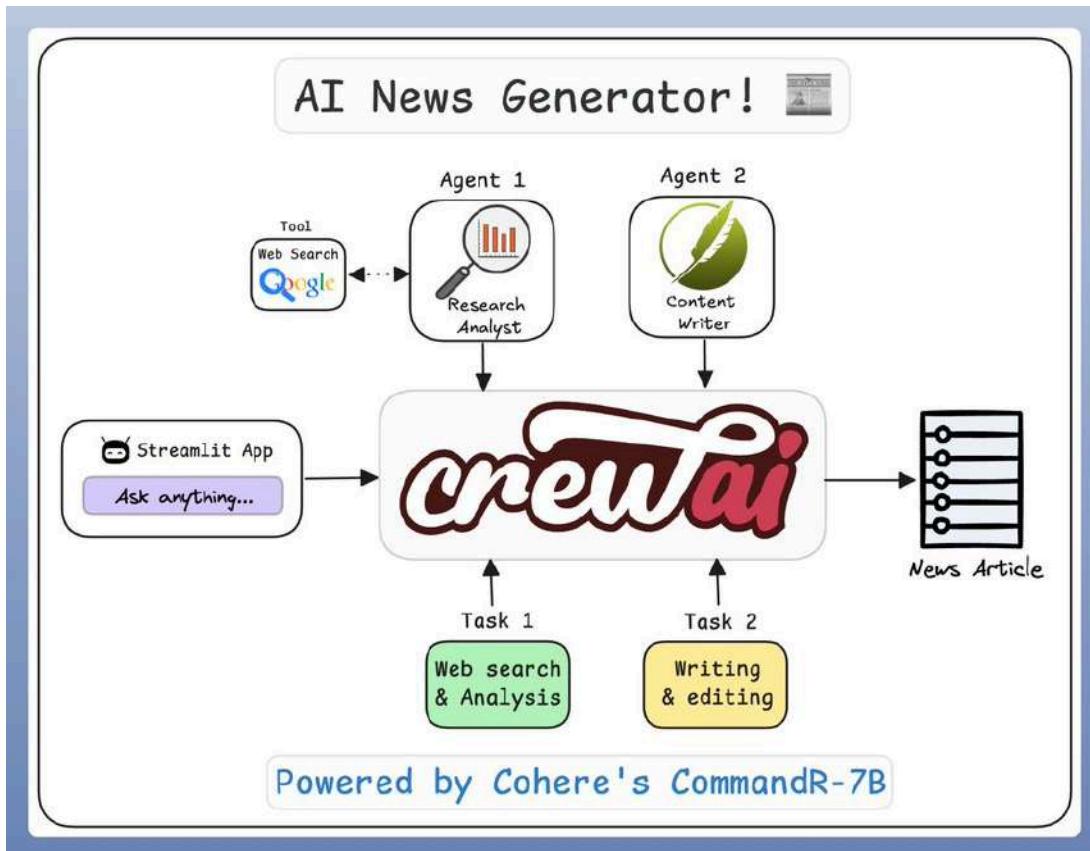
# Cloning repository: https://github.com/nikhil-teo/machine-learning-daily

Cloning into 'workdir/machine-learning-daily'...
remote: Enumerating objects: 621, done.
remote: Counting objects: 100% (621/621), done.
remote: Compressing objects: 100% (516/516), done.
remote: Total 621 (delta 135), reused 570 (delta 87), pack-reused 0 (from 0)
Receiving objects: 100% (621/621), 1.77 MiB | 2.32 MiB/s, done.
Resolving deltas: 100% (135/135), done.
Flow: CreateDocumentationFlow
ID: 0f54b988-af69-4dfb-9433-158f3feeb42f
└ Flow Method Step
  └ completed: clone_repo

Flow: CreateDocumentationFlow
ID: 0f54b988-af69-4dfb-9433-158f3feeb42f
└ Flow Method Step
  └ completed: clone_repo
  └ running: plan_docs
```

## #12) News Generator

The app takes a user query, searches the web for it, and turns it into a well-crafted news article, with citations.



Tech stack:

- Cohere ultra-fast Command R 7B as LLM
- CrewAI for multi-agent orchestration

Workflow:

We'll have two agents in this multi-agent app:

Research analyst agent:

- Accepts a user query.
- Uses the Serper web search tool to fetch results from the internet.
- Consolidates the results.

Content writer agent:

- Uses the curated results to prepare a polished, publication-ready article.

Let's implement this:

### #1) Setup

Create a .env file for their corresponding API keys:

- Cohere API key
- Serper API key

Next, setup the LLM and web search tool as follows:

The image shows a terminal window with two tabs. The top tab is titled "env-vars" and contains the following environment variables:

```
SERPERS_API_KEY=your_serper_api_key  
COHERE_API_KEY=your_cohere_api_key
```

The bottom tab shows Python code for initializing an LLM and a search tool:

```
from crewai import LLM  
from crewai_tools import SerperDevTool  
  
llm = LLM(  
    model="command-r7b-12-2024",  
    temperature=0.7  
)  
  
search_tool = SerperDevTool(n_results=10)
```

### #2) Senior Research Analyst Agent

The web-search agent takes a user query and then uses the Serper web search tool to fetch results from the internet and consolidate them. Check this out:



```
from crewai import Agent

senior_research_analyst = Agent(
    role="Senior Research Analyst",
    goal="Research, analyze, and synthesize comprehensive information on {topic} from \n"
        "reliable web sources",
    backstory="You're an expert research analyst with advanced web research skills. "
        "You excel at finding, analyzing, and synthesizing information from "
        "across the internet using search tools. You're skilled at "
        "distinguishing reliable sources from unreliable ones, "
        "fact-checking, cross-referencing information, and "
        "identifying key patterns and insights. You provide "
        "well-organized research briefs with proper citations "
        "and source verification. Your analysis includes both "
        "raw data and interpreted insights, making complex "
        "information accessible and actionable.",
    allow_delegation=False,
    verbose=True,
    tools=[search_tool],
    llm=llm
)
```

This is the research task that we assign to our senior research analyst agent, with description and expected output.

```
from crewai import Task

research_task = Task(
    description=""""
        1. Conduct comprehensive research on {topic} including:
            - Recent developments and news
            - Key industry trends and innovations
            - Expert opinions and analyses
            - Statistical data and market insights
        2. Evaluate source credibility and fact-check all information
        3. Organize findings into a structured research brief
        4. Include all relevant citations and sources
    """),
    expected_output="""A detailed research report containing:
        - Executive summary of key findings
        - Comprehensive analysis of current trends and developments
        - List of verified facts and statistics
        - All citations and links to original sources
        - Clear categorization of main themes and patterns
        Please format with clear sections and bullet points for easy reference."""
)
agent=senior_research_analyst
)
```

### #3) Content writer agent

The role of content writer is to use the curated results and turn them into a polished, publication-ready news article .



```
from crewai import Agent

# Second Agent: Content Writer
content_writer = Agent(
    role="Content Writer",
    goal="Transform research findings into engaging blog posts while maintaining accuracy",
    backstory="You're a skilled content writer specialized in creating "
              "engaging, accessible content from technical research. "
              "You work closely with the Senior Research Analyst and excel at maintaining \n"
              "the perfect balance between informative and entertaining writing, "
              "while ensuring all facts and citations from the research "
              "are properly incorporated. You have a talent for making "
              "complex topics approachable without oversimplifying them.",
    allow_delegation=False,
    verbose=True,
    llm=llm
)
```

This is how we describe the writing task with all the details and expected output:

```
from crewai import Task

# Writing Task
writing_task = Task(
    description=""""
        Using the research brief provided, create an engaging blog post that:
        1. Transforms technical information into accessible content
        2. Maintains all factual accuracy and citations from the research
        3. Includes:
            - Attention-grabbing introduction
            - Well-structured body sections with clear headings
            - Compelling conclusion
        4. Preserves all source citations in [Source: URL] format
        5. Includes a References section at the end
    """,
    expected_output=""""
        A polished blog post in markdown format that:
        - Engages readers while maintaining accuracy
        - Contains properly structured sections
        - Includes Inline citations hyperlinked to the original source url
        - Presents information in an accessible yet informative way
        - Follows proper markdown formatting, use H1 for the title and H3 for the sub-sections"",
    agent=content_writer
)
```

#### #4) Setup Crew

And we're done!

Just build a crew and kick it off!

